



Matrix Query Language (MQL) Guide

Version 10

Copyright and Trademark Information

© Copyright 1994–2003 by MatrixOne Inc.

All rights reserved.

PROPRIETARY RIGHTS NOTICE: This documentation is proprietary property of MatrixOne, Inc. In accordance with the terms and conditions of the Software License Agreement between the Customer and MatrixOne, the Customer is allowed to print as many copies as necessary of documentation copyrighted by MatrixOne relating to the eMatrix software being used. This documentation shall be treated as confidential information and should be used only by employees or contractors with the Customer in accordance with the Agreement.

MatrixOne®, eMatrix®, and Adaplet® are registered trademarks of MatrixOne, Inc.

Value Chain Portfolio, AEF, Application Exchange Framework, MatrixOne Document Central, MatrixOne Engineering Central, MatrixOne Program Central, MatrixOne Sourcing Central, MatrixOne Supplier Central, MatrixOne Team Central, IconMail, ImageIcon, Primary Browser, Star Browser, and State Browser are trademarks of MatrixOne Inc. Oracle® is a registered trademark of Oracle Corporation, Redwood City, California. DB2, AIX, and WebSphere are registered trademarks of IBM Corporation. WebLogic is a registered trademark of BEA Systems, Inc. Solaris, Sun ONE, UltraSPARC, Java, JavaServer Pages, JDBC, and J2EE are registered trademarks of Sun Microsystems, Inc. Windows 98, Windows NT, Windows 2000, and Windows XP, and Internet Explorer are registered trademarks of Microsoft Corp. HP and HP-UX are registered trademarks of HP. Orbix® is a registered trademark of IONA Technologies PLC, Dublin, Ireland. CORBA®, Object Request Broker™, ORB™, and IIOP™ are trademarks or registered trademarks of Object Management Group, Inc. in the U.S. and in other countries. All other product names and services identified throughout this book are recognized as trademarks, registered trademarks, or service marks of their respective companies.

This product includes software developed by the Apache Software Foundation. (<http://www.apache.org/>)

This product includes software developed by the OpenLDAP™ Project for use in the openLDAP Toolkit. (<http://www.openldap.org/>)

This product includes software developed by the OpenSSL Project for use in the OpenSSL Toolkit. (<http://www.openssl.org/>)

This product includes cryptographic software written by Eric Young. (eay@cryptsoft.com)

This product includes GifEncoder and ImageEncoder software written by Jef Poskanzer (jef@acme.com). Copyright ©1996,1998. All rights reserved.

MatrixOne Inc.
210 Littleton Road
Westford, MA 01886, USA
Telephone: 978-589 4000
Fax: 978-589-5700
Email: info@matrixone.com

Web Address: <http://www.matrixone.com>

DM-MX-06-10-00

Table of Contents

Preface..... xxv

Part I: General Functions

Chapter 1. Introduction to MQL	33
MQL Defined.....	35
Three Modes: Interactive, Script, and Tcl.....	36
Script Mode Advantages.....	36
Using MQL.....	36
Command Line Editing.....	37
Tcl/Tk Mode	39
Tcl Versions	41
MQL and Tcl	42
Accessing MQL.....	43
MQL Command Options	44
-b Option	44
-c Option	45
-d Option	45
-k Option	45
-t Option	45
-v Option	45
-stdout:FILENAME	45
-stdin:FILENAME	45
-stderr:FILENAME	46
Statements and Scripts.....	47
MQL Statements.....	48
Entering (Writing) MQL Statements	48
Reviewing Statements	49
Important eMatrix Statements.....	50
Building an MQL Script	53
Running Scripts and Using Basic MQL Statements.....	53
Using Comments	54
Building an Initial eMatrix Database.....	54
Modifying an Existing Database.....	56
General Syntax	57
Add Statement	57
Copy Statement	57
Modify Statement	58
List Statement	58
List Admintype Statement	59
Print Statement	61
Delete Statement	62
Help Statement	62
Icon Clause	63

In Vault Clause	63
Evaluating Expressions.....	64
Business Object Collection Clause	64
Dump Clause	67
Recordseparator Clause	67
Examples	67
Tcl Format Select Output.....	68
Chapter 2. Working With Transactions	73
Implicit and Explicit Transactions.....	74
Implicit Transactions	74
Explicit Transaction Control	74
Working With Threads	77
Start Thread Command.....	77
Resume Thread Command.....	77
Print Thread Command.....	77
Kill Thread Command	77

Part II: System Administrator Functions

Chapter 3. Maintaining the System	81
System Administrator Responsibilities	82
Controlling System-wide Settings	83
Updating Sets With Change Vault.....	83
Database Constraints.....	84
System Decimal Symbol	85
Allowing Empty Strings for Object Names	85
Setting History Logging for the System.....	86
Privileged Business Administrators.....	86
System tidy.....	86
List Statement	87
Print Statement	87
Validating the eMatrix Database	88
Using an Oracle Database	89
Using a DB2 Database.....	89
Maintaining and Monitoring Clients	90
Viewing User Session Information	90
Error Log and Disk Space	90
Access Log.....	90
Monitoring Servers	95
Enabling Tracing	95
Enabling trace tools via MQL	97
ADK interfaces	98
Output	100
Tracing and Debugging.....	101
Examples	102
ADK Session Monitoring	105
Developing a Backup Strategy.....	110

Chapter 4. Working With Vaults	111
Vault Defined	112
Kinds of Vaults	113
Business Object Vaults	113
Administration Vault	113
Defining a Vault	114
Description Clause.....	114
Icon Clause	115
Vault Types	115
Tablespace Clause	116
Indexspace Clause	116
Server Clause	116
Interface Clause	116
File Clause	116
Map Clause	117
Hidden Clause	117
Property Clause	117
Modifying a Vault Definition	118
Modifying a Vault Definition	118
Clearing and Maintaining Vaults	119
Clearing Vaults.....	119
Indexing Vaults	119
Fixing Fragmented Vaults	120
Updating Sets With Change Vault	120
Printing a Vault Definition.....	121
Deleting a Vault	122
Chapter 5. Working With Stores	123
Store Defined.....	124
Types of File Stores	125
Ingested Files.....	125
Captured Files.....	125
Tracked Files	126
Defining a Store	127
Description Clause.....	128
Icon Clause	128
Type Clause	129
Filename Hashed Clause.....	129
Lock Clause	131
Path Clause	131
Host Clause	131
Protocol and Port Clauses	132
Permission Clause	133
User and Password Clauses.....	134
URL Clause.....	134
Tablespace Clause	135
Indexspace Clause	135
Hidden Clause	135
Property Clause	135
Enabling Secure FTP for a Captured Store	136

Modifying Store Definitions	138
Modifying a File Store	138
Maintaining a File Store	140
Tidy Store Statement	140
Inventory Store Statement	140
Rehash Store Statement.....	141
Validate Store Statement	142
Monitoring Disk Space	142
Deleting a Store	144
Purging Files	144
Chapter 6. Replicating Captured Stores	147
Captured Store Replication	148
Implications of Changing Stores	149
Defining a Location	150
Description Clause	150
Host Clause.....	150
Icon Clause	151
Protocol and Port Clauses	151
Password Clause	151
Path Clause.....	152
Permission Clause	152
User Clause	153
URL Clause.....	153
Hidden Clause.....	154
Property Clause	154
Modifying a Location Definition	155
Modifying a Location Definition	155
Deleting a Location	157
Purging Files	157
Defining Sites	158
Description Clause	158
Icon Clause	158
Member Clause..	159
Hidden Clause.....	159
Property Clause	159
Modifying a Site Definition	160
Modifying a Site Definition.....	160
Deleting a Site.....	161
Synchronizing Captured Stores	162
Automatic Synchronization.....	162
Manual Synchronization	162
Using format.file	165
Tidying all locations.....	165
Chapter 7. Distributing the Database	167
Distributed and Replicated Data	168
Preparing for Distribution	169
System Setup and Installation.....	169
Working with Servers	171

Defining a Server	171
Description Clause	172
Icon Clause	172
User Clause	172
Password Clause	172
Connect Clause	172
Timezone Clause	172
Hidden Clause	175
Property Clause	175
Modifying a Server Definition	176
Modifying a Server Definition	176
Disabling or Deleting a Server	177
Disabling a Server.....	177
Deleting a Server	177
Distributing Data Across Servers.....	178
Setting Up a Distributed Database.....	178
Considerations	180
Network Stability	180
Locating the Master Vault	180
Renaming Servers	180
Distribution Problems.....	180
Importing Servers.....	181
Should I Use a Link or a Copy?	181
Using Schema Names	183
Requirements.....	183
Use Case	183
Chapter 8. Working with Indices.....	185
Index Defined.....	186
Defining an Index	187
Description Clause.....	187
Icon Clause	187
Attribute Clause	188
Hidden Clause	188
Property Clause	188
Modifying an Index Definition.....	189
Enabling/Disabling an Index	190
Improving Performance with Indices.....	191
Validating an Index	191
Using the Index as Select Output	191
Deleting an Index	192
Chapter 9. Working With Export/Import.....	193
Overview of Export/Import	195
Exporting Administrative Objects	196
Export statement.....	196
ADMIN_TYPE Clause.....	196
OPTION_ITEMS	197
XML Clause	198
Into File/Onto File.....	198

Exclude Clause	198
Log FILENAME Clause	198
Exception FILENAME Clause	199
Exporting Business Objects	200
Export Bus Statement	200
OPTION_ITEMS	201
Exporting Workflows	203
Export Workflow Statement.....	203
Exporting Objects to XML	204
XML Export Requirements and Options.....	204
XML Clause.....	205
XML Statement	205
XML Output	205
Importing Administrative Objects	209
Import Statement.....	209
List Clause	210
Admin and ADMIN_TYPE Clauses	210
OPTION_ITEMS	211
Use FILE_TYPE Clauses.....	212
Importing Servers.....	213
Importing Workspaces	213
Importing Properties.....	214
Importing Business Objects	216
Import Bus Statement	216
List Clause	217
OPTION_ITEMS	217
Use FILE_TYPE Clauses.....	219
Importing Workflows	221
Import Workflow Statement.....	221
Extracting from Export Files.....	222
OPTION_ITEMS	222
Examples	222
Migrating Databases	223
Migrating Files.....	223
Comparing Schema	225
Scenario	225
Creating a Baseline.....	225
Comparing a Schema with the Baseline	226
Reading the Report	229
Examples	232

Part III: Business Administrator Functions

Chapter 10. User Access	241
Administrative Objects that Control Access	242
Persons	242
User Categories	243
Policies.....	244
Rules.....	245
Working With Expression Access Filters.....	245

Access that is Granted.....	247
Which Access Takes Precedence Over the Other?.....	248
Access Precedence: Description	248
Access Precedence: Flow Chart.....	250
Accesses	251
Summary of All Accesses	251
More About Read Access	255
More About Unlock Access	256
More About Show Access	256
More about Connection Accesses	258
Summary of User Access Goals	259
Chapter 11. Working With Users	261
Kinds of Users	262
Integrating with LDAP and Third-Party Authentication Tools.....	263
System-Wide Password Settings.....	264
Encrypting Passwords	267
Working with Persons	268
Defining a Person	268
Copying and/or Modifying a Person Definition.....	284
Copying (Cloning) a Person Definition.....	284
Modifying a Person Definition	284
Deleting a Person	287
Determining When to Create a Group, Role, or Association	288
Working with Groups and Roles	290
Establishing Group and Role Hierarchy.....	290
Defining a Group or Role	291
Modifying a Group or Role Definition	295
Deleting a Group or Role Definition	295
Working with Associations	298
Uses for Associations	298
Defining an Association	299
Copying (Cloning) an Association Definition.....	302
Modifying an Association Definition	302
Deleting an Association	303
Role-Based Visuals	304
Sharing Visuals	304
Chapter 12. Working With Attributes	307
Assigning Attributes to Objects and Relationships	308
Assigning Attributes to Objects	308
Assigning Attributes to Relationships.....	308
Defining an Attribute	309
Type Clause	309
Default Clause	311
Description Clause	311
Icon Clause	311
Rule Clause	312
Range Clause	312

Trigger Clause.....	317
Multiline Clause.....	317
Hidden Clause.....	317
Property Clause	317
Checking an Attribute.....	319
Copying and/or Modifying an Attribute Definition.....	320
Copying (Cloning) an Attribute Definition	320
Modifying an Attribute Definition	320
Deleting an Attribute	322
Chapter 13. Working With Types	323
Type Defined.....	324
Type Characteristics	325
Implicit and Explicit.....	325
Inherited Properties.....	325
Defining a Type	326
Description Clause	326
Icon Clause	327
Attribute Clause.....	327
Derived Clause.....	328
Abstract Clause.....	329
Method Clause	329
Form Clause.....	329
Trigger Clause.....	330
Hidden Clause.....	330
Property Clause	331
Copying and/or Modifying a Type Definition	332
Copying (Cloning) a Type Definition	332
Modifying a Type Definition	332
Three Base Types.....	334
Localizing Base Types	334
Deleting a Type	335
Chapter 14. Working With Relationships	337
Overview of Relationships	338
Collecting Data for Defining Relationships.....	339
Defining a Relationship	341
Attribute Clause.....	341
Trigger Clause.....	342
Description Clause	343
Icon Clause	343
To and From Clauses.....	344
PreventDuplicates Clause	352
Hidden Clause.....	353
Property Clause	353
Copying and/or Modifying a Relationship Definition	354
Copying (Cloning) a Relationship Definition.....	354
Modifying a Relationship Definition	354
Connection End Modifications.....	356
Deleting a Relationship	358

Working with Relationship Instances	359
Modifying Relationships	359
Connection IDs	359
Modifying Attributes of a Connection	360
Freezing Connections	360
Thawing Connections	360
Deleting Connections.....	360
Printing Connections.....	360
Chapter 15. Working With Formats	363
Overview of Formats.....	364
Defining a Format	365
Description Clause.....	365
Creator and Type Clauses	366
View, Edit, and Print Clauses.....	366
Icon Clause	367
Suffix Clause.....	367
Mime Clause	368
Version Clause.....	368
Hidden Clause	368
Property Clause	369
Copying and/or Modifying a Format Definition.....	370
Copying (Cloning) a Format Definition.....	370
Modifying Format Definitions	370
Deleting a Format	372
Chapter 16. Working With Rules.....	373
Rule Defined	374
Creating Rules	375
Creating a Rule	375
Description Clause.....	375
Icon Clause	375
Hidden Clause	376
Property Clause	376
Assigning Access.....	376
Copying and/or Modifying a Rule Definition.....	379
Copying (Cloning) a Rule Definition.....	379
Modifying a Rule Definition	379
Assigning Rules	381
Deleting a Rule	382
Chapter 17. Working With Policies.....	383
Policy Defined	384
General Behavior	384
Lifecycle	384
Determining Policy States.....	385
How Many States are Required?	385
Who Will Have Object Access?	386
Is the Object Versionable or Revisionable?	386

How Do You Change From One State to the Next?	386
Defining an Object Policy.....	388
Description Clause	388
Icon Clause	389
Type Clause	389
Format Clause.....	390
Defaultformat Clause	391
Sequence Clause.....	391
Enforce Clause.....	393
State Clause.....	394
Store Clause	405
Hidden Clause.....	406
Property Clause	406
Copying and/or Modifying a Policy Definition.....	408
Copying (Cloning) a Policy Definition.....	408
Modifying a Policy Definition	408
Modifying Policy States	411
Modifying Signature Requirements	413
Deleting a Policy	415
Chapter 18. Working With Business Wizards.....	417
Overview of Business Wizards	418
What is a Frame?.....	418
What is a Widget?	419
Runtime Program Environment.....	420
Using \${} Macros.....	421
Planning the Wizard	421
Creating a Business Wizard.....	422
Code Clause	422
Description Clause	423
Wizard Program Type (External or MQL).....	423
File Clause	423
Icon Clause	423
Needs Business Object Clause.....	423
Downloadable Clause	424
Execute Clause	424
Hidden Clause.....	425
Property Clause	425
Frame Clause.....	425
Programming Business Wizards	436
Strategy for Creating Business Wizards	436
Program Environment Variables	437
Loading Complex Widgets	440
Using Spaces in Strings	440
Using \${} Macros.....	440
Using Eval Statements	443
MQL Download/Upload Commands	444
Download Command.....	444
Upload Command	444
Upload Command vs. Widget Upload Option	445

Running/Testing the Business Wizard	447
.Command Line Interface.....	447
eMatrix Interface	447
Invoking a Wizard from a Program	448
Copying and/or Modifying a Business Wizard Definition	450
Copying (Cloning) a Wizard Definition	450
Modifying a Wizard Definition	450
Modifying Wizard Frames	452
Modifying Widgets.....	452
Deleting a Business Wizard.....	454
Chapter 19. Working With Programs.....	455
Overview of Programs	457
Creating a Program	458
Code Clause	458
Description Clause.....	461
Java or MQL or External Clause.....	461
File Clause	462
Rule Clause	462
Icon Clause	462
Needs Business Object Clause	462
Downloadable Clause	463
Execute Clause.....	463
Hidden Clause	464
Piped Clause.....	464
Pooled Clause.....	465
Property Clause	465
Handling Variables.....	467
Using the Runtime Program Environment	467
Macro Processing and Command Syntax.....	470
Programs for Attribute Ranges	472
Transaction Management.....	473
Transaction Boundaries	473
Java Program Objects	474
Java Command.....	474
Compile command	475
Testing Timing and Debugging Programs	476
Program Strategies	477
Nesting Programs	477
Notifying Users (Notice, Warning and Error Commands)	477
Writing Piped Program Code	478
Program Object Select Expressions	481
Program With Arguments.....	482
Programming for Use in a Loosely-Coupled Database.....	484
Programs Exported to XML.....	485
Programs Using Output Length	485
Optimizing Programs	486
Access Business Objects and Relationships by Vault or OID	486
Selecting Relationship and Business Object Data.....	487
Referencing Schema Object Names.....	488

Query Performance	488
Avoid Initializing multiple Tcl Interpreters	488
Using Transaction Boundaries	489
Storing "Static" Data	489
Program Exit vs. Program Failure	490
Generating an Exit Code	491
Program Failure	492
Dialog Scripting Language	493
Application Command	493
Application Help	497
Using the Application Command	497
Programming for the Web Version	500
Downloadable Programs	500
Downloadable Program Threads Share Context	500
Tcl Programs	503
File Checkin and Checkout	503
Dialog Scripting on the Web	504
Web URL Access	504
Executing a Program	505
Command Line Execution	505
Dealing With User Context	506
Push Context Command	507
Pop Context Command	507
Initialization File Context Variable	507
Program Output	508
Copying and/or Modifying a Program Definition	510
Copying (Cloning) a Program Definition	510
Modifying a Program Definition	510
Deleting a Program	513
Chapter 20. Working With Event Triggers.....	515
Overview of Event Triggers	516
Trigger Scenarios	517
Transaction Boundaries and Events	518
Transactions Boundaries and Action Triggers	518
Designing Triggers	520
Supported Events	520
Types of Triggers	521
Multi-Trigger Events	522
User Interaction	524
Lifecycle Checks and Actions with Event Triggers	525
Trigger Programs	526
Trigger Macros	526
Check Programs	526
Override Programs	527
Action Programs	528
Validate Query Trigger	529
Enabling the Validate Query Trigger	529
Using the ValidateQuery Trigger with matrixcgi	530
Writing the ValidateQuery Program	530

Use Case	531
Recursion Detection Modes and Limits	535
Limiting the Trigger Stack Depth.....	536
Override Example	536
Logging Trigger Events.....	539
Enable/Disable Triggers	540
Chapter 21. Working With Workflow Processes	541
Overview of Workflow Processes	542
Defining a Process.....	543
Description Clause.....	543
Attribute Clause	544
Interactive Clause	544
Automated Clause	548
Subprocess Clause.....	551
And Clause	552
Or Clause.....	552
Finish Clause	552
Start Clause	552
Stop Clause	553
Autostart Clause	553
Hidden Clause	553
Icon Clause	553
Property Clause	553
Trigger Clause	554
Copying and/or Modifying a Process Definition	555
Copying (Cloning) a Process Definition	555
Modifying a Process Definition.....	555
Reassigning an Activity to a Group.....	558
Using Links	559
Connect Clause	559
Disconnect Clause	559
Transition Conditions	559
Deleting a Process.....	561
Validating a Process	562
Chapter 22. Working With Reports	563
Overview of Reports	564
Designing the Definition and Layout of a Report	565
Defining a Report	566
Units Clause.....	566
Description Clause.....	567
Icon Clause	568
Size Clause.....	568
Header Clause	568
Footer Clause	569
Margins Clause	570
Displayrule Clause	571
Field Clause	571
Hidden Clause	580

Property Clause	580
Evaluating a Report	581
Copying and/or Modifying a Report	582
Copying (Cloning) a Report Definition.....	582
Modifying a Report	582
Deleting a Report	584
Chapter 23. Working With Forms.....	585
Overview of Forms	586
Defining a Form	587
Units Clause.....	587
Description Clause	588
Icon Clause	589
Rule Clause.....	589
Color Clause	589
Header Clause	589
Footer Clause.....	590
Margins Clause	590
Type Clause	591
Size Clause	591
Field Clause	591
Hidden Clause.....	595
Property Clause	595
Copying and/or Modifying a Form	597
Copying (Cloning) a Form Definition.....	597
Modifying a Form	597
Deleting a Form	599
Printing a Form	600
Chapter 24. Working With Administration Properties	603
Overview of Administration Properties	604
Defining a Property	605
Adding Properties to Administrative Definitions	606
Adding Properties to User Workspace Items	606
Modifying Properties	606
Deleting Properties.....	606
Listing Properties	606
Selecting Properties	607
Chapter 25. Working With Aliases.....	609
Aliases Defined	610
Alias Properties	610
What Remains in Base Language.....	610
Enabling Aliases.....	611
Working with Language Aliases.....	612
Add Statement	612
Modify Statement	612
Delete Statement	612
Overriding the Language.....	613

Implementation Issues	613
Chapter 26. Working With Pages	615
Overview.....	616
Defining a Page Object.....	617
Content Clause	617
Description Clause.....	618
File Clause	618
Icon Clause	618
Mime Clause	619
Hidden Clause	619
Property Clause	619
Copying (Cloning) a Page Definition.....	620
Modifying a Page Definition	621
Deleting a Page	622
Supporting Alternate Languages and Display Formats	623
Chapter 27. Working With Resources	625
Overview.....	626
Defining a Resource	627
Description Clause.....	627
File Clause	627
Mime Clause	628
Icon Clause	628
Hidden Clause	628
Property Clause	628
Copying (Cloning) a Resource Definition.....	629
Modifying a Resource Definition	630
Deleting a Resource	631
Supporting Alternate Languages and Display Formats	632
Chapter 28. Working With Commands	633
Commands Defined	634
Creating a Command.....	635
Description Clause.....	635
Icon Clause	636
Label Clause	636
Href Clause	636
Alt Clause.....	636
Code Clause	637
File Clause	637
User Clause	637
Setting Clause.....	637
Property Clause	638
Copying and/or Modifying a Command	639
Copying (Cloning) a Command Definition.....	639
Modifying a Command	639
Deleting a Command	641
Using Macros and Expressions in Configurable Components.....	642

Supported Macros and Selects	643
Chapter 29. Working With Menus	645
Menus Defined.....	646
Creating a Menu	647
Description Clause	647
Icon Clause	647
Label Clause	648
Href Clause	648
Alt Clause	648
Menu Clause	648
Command Clause	649
Setting Clause.....	649
Property Clause	649
Copying and/or Modifying a Menu	650
Copying (Cloning) a Menu Definition	650
Modifying a Menu.....	650
Deleting a Menu.....	653
Chapter 30. Working With Inquiries.....	655
Inquiries Defined	656
Creating an Inquiry.....	657
Description Clause	657
Icon Clause	657
Pattern Clause	657
Format Clause.....	658
Code Clause	658
File Clause	659
Argument Clause	659
Property Clause	659
Copying and/or Modifying an Inquiry	660
Copying (Cloning) an Inquiry Definition.....	660
Modifying an Inquiry	660
Evaluating an Inquiry	662
Deleting a Inquiry	663
Part IV: eMatrix User Functions	
Chapter 31. Working With Context	667
Context Defined	668
Setting Context	669
Setting Context With Passwords	669
Setting Context Without Passwords	670
Setting Context With Disabled Passwords	671
Setting Context Temporarily.....	671
Chapter 32. Working With IconMail	673
Overview of IconMail.....	674

Sending IconMail	675
To Clause.....	675
CC Clause.....	676
Subject Clause	676
Text Clause.....	676
Reading IconMail	677
Deleting an IconMail Message.....	678
Deprecated Function.....	678
Chapter 33. Working With Workflows	679
Overview of Workflows	680
Defining a Workflow	681
Description Clause.....	681
Image Clause.....	681
Vault Clause.....	683
Owner Clause	683
Attribute Clause	683
Managing Workflows	685
Starting a Workflow.....	685
Stopping a Workflow	685
Suspending a Workflow	685
Resuming a Workflow	685
Reassigning a Workflow	686
Planning Workflow Execution	686
Modifying a Workflow Definition.....	688
Modifying a Workflow Definition.....	688
Assigning Ad hoc Workflow Activity to Multiple Users	689
Deleting a Workflow	691
Chapter 34. Creating and Modifying Business Objects	693
Business Objects	694
Specifying a Business Object Name.....	695
Business Object Type	695
Business Object Name	695
Business Object Revision Designator.....	696
Object ID	697
Adding a Business Object.....	698
Description Clause	698
Image Clause.....	699
Vault Clause.....	700
Revision Clause	701
Owner Clause	701
Policy Clause	701
State Clause	702
Attribute Clause	703
Viewing Business Object Definitions	705
Print Businessobject Statement.....	705
Select Statements.....	706
Using select format.file.path.....	710
Copying and Modifying a Business Object	711

Copying/Cloning a Business Object.....	711
Modifying a Business Object.....	711
Creating a Revision of an Existing Business Object.....	718
Adding an Object to a Revision Sequence.....	719
Deleting a Business Object and Its Files	721
Chapter 35. Working with Business Objects	723
Making Connections Between Business Objects.....	724
Connect Businessobject Statement	724
Disconnect Businessobject Statement.....	725
Modify Connection Statement	726
Displaying and Searching Business Object Connections	728
From Clause.....	729
To Clause.....	729
Relationship Clause	730
Type Clause	730
Filter and Activefilter Clauses.....	731
Recurse Clause.....	731
Set Clause.....	732
Structure Clause	732
SELECT_BO and SELECT_REL Clauses	733
Dump Clause and Output Clause	734
Tcl Clause	736
Recordseparator Clause	736
Terse Clause.....	736
Limit Clause	736
Working with Saved Structures.....	737
Checking Files In and Out of a Business Object.....	738
Checking In Files.....	738
Checking Out Files.....	741
Moving Files	743
Locking and Unlocking a Business Object.....	744
Locking a Business Object.....	744
Unlocking a Business Object	745
Modifying the State of a Business Object	746
Approve Businessobject Statement	746
Ignore Businessobject Statement	747
Reject Businessobject Statement	747
Unsign Signature.....	748
Disable Businessobject Statement.....	748
Enable Businessobject Statement	748
Override Businessobject Statement.....	749
Promote Businessobject Statement	750
Demote Businessobject Statement.....	750
Chapter 36. Working With History	751
History Overview.....	752
Adding a Custom History Record for Business Objects.....	753
Selecting History Entries	754
Excluding History when Printing or Expanding	754

Selecting History	754
Deleting History	758
delete history clause	758
Important Notes	762
Enable/Disable History	763
Chapter 37. Working With Sets	765
Set Defined	766
Understanding Differences	766
Creating a Set	767
Copying or Modifying a Set Definition.....	770
Copying (Cloning) a Set Definition.....	770
Modifying a Set Definition	770
Expanding Objects Within Sets	772
Relationship Expressions.....	772
Expanding From.....	772
Expanding To.....	773
Relationship Clause.....	774
Type Clause	774
Recurse Clause	775
Set Clause	775
Dump and Output Clauses.....	776
Tcl Clause	776
Terse Clause.....	776
Viewing Set Definitions	777
Select Clause.....	777
Dump Clause	778
Recordseparator Clause	778
Tcl Clause	778
Output Clause	779
Expressions on Sets	780
Count	780
Sum.....	780
Maximum, Minimum, Average	781
Median	781
Standard Deviation	781
Correlation	782
Deleting a Set	783
Chapter 38. Working With Queries	785
Query Overview	786
Performing a Temporary Query	787
Defining a Saved Query.....	789
Businessobject Clause.....	790
Hidden Clause	791
Owner Clause	791
Vault Clause.....	791
Expandtype Clause.....	791
Where Clause	792
Using Select Clauses in Queries	801

Using Fromset and Toset Selectables	805
Query Strategies	808
Using Select Fields	808
Using Wildcard Characters in a Where Clause.....	810
Searching based on lengthy string fields	811
Modeling Considerations.....	811
Searching on Date/Time	811
Using Parentheses.....	812
Using const for reserved words	812
Expressions on Queries	813
Using the Escape Character	817
To Enable Escaping	817
How It Works	818
Using Escaping With Tcl	819
Exceptions.....	819
Modifying a Query Definition.....	820
Evaluating Queries.....	821
Deleting a Query	823
Chapter 39. Working With Filters.....	825
Filter Defined.....	826
Creating Filters.....	827
Active Clause	827
Applies to Clause	827
Direction Clause.....	827
Type Clause	828
Name Clause	828
Revision Clause	828
Vault Clause	829
Owner Clause	829
Where Clause	829
Hidden Clause.....	830
Property Clause	830
Copying and/or Modifying a Filter	831
Copying (Cloning) a Filter Definition	831
Modifying a Filter.....	831
Deleting a Filter	833
Chapter 40. Working With Cues.....	835
Cues Defined	836
Creating a Cue	837
Adding a Cue	837
Active Clause	838
Appliesto Clause	838
Type Clause	838
Name Clause	838
Revision Clause	838
Color, Font, Highlight, and Linestyle Clauses	839
Vault Clause	839
Order Clause.....	839

Owner Clause	840
Where Clause	840
Hidden Clause	840
Property Clause	840
Copying or Modifying a Cue	841
Copying (Cloning) a Cue Definition.....	841
Modifying a Cue	841
Deleting a Cue	844
Chapter 41. Working With Tips	845
Object Tips Defined	846
Creating a Tip	847
Active Clause	847
Applies To Clause.....	847
Type Clause.....	847
Name Clause	848
Revision Clause	848
Vault Clause.....	848
Owner Clause	849
Where Clause	849
Expression Clause	849
Hidden Clause	849
Property Clause	850
Copying and/or Modifying a Tip	851
Copying (Cloning) a Role Definition.....	851
Modifying a Tip.....	851
Deleting a Tip.....	853
Chapter 42. Working With Toolsets	855
Toolsets Defined.....	856
Creating Toolsets.....	857
Active Clause	857
Program Clause	857
Method Clause	857
Hidden Clause	857
Property Clause	857
Copying and/or Modifying a Toolset	859
Copying (Cloning) a Toolset Definition	859
Modifying a Toolset Definition.....	859
Deleting a Toolset.....	861
Chapter 43. Working With Tables	863
Tables Defined.....	864
Creating a Table	865
Active Clause	865
Units Clause.....	865
Description Clause.....	866
Icon Clause	867
Column Clause	867

Hidden Clause.....	869
Property Clause	869
Copying and/or Modifying a Table	870
Copying (Cloning) a Table Definition	870
Modifying a Table.....	870
Evaluating a Table	873
Using Collections	873
Deleting a Table	875
Printing a Table	876
Chapter 44. Working With Views	879
Views Defined	880
Creating a View.....	881
Active Clause	881
Hidden Clause.....	881
Property Clause	881
Copying a View	883
Setting the Workspace	884
Modifying a View	885
Deleting a View	887
Macros Appendix.....	889
Using Macros	890
Macro Categories.....	892
Intrinsic Macros	892
Business Object Identification Macros	893
Format Macros	895
Lifecycle Check and Action Macros	895
Wizard Macros	895
Program Range Macros	896
Dynamic UI Macros	897
Connection Macros	898
Workflow Macros.....	899
JPO Macros	899
Trigger Macros	901
Trigger Macro Categories	901
Trigger Event Macros.....	906
Attribute Event Macros	906
Business Object Event Macros	906
Connection Event Macros	909
State Event Macros.....	910
Process Event Macros	912
Activity Event Macros.....	913
Query Event Macros	914
Macro Processing and Command Syntax.....	915
Index	917

Preface

The *Matrix Query Language (MQL) Guide* is designed as a reference tool for eMatrix Business and System Administrators.

Before You Begin...

Before you begin administrative work, you should review the following information:

- *eMatrix Basics*
Review this entire guide. It presents eMatrix concepts and lists common terminology.
- *eMatrix Navigator Guide*
You should become familiar with all of the information in the *eMatrix Navigator Guide* to review the basic skills that a user will need for using eMatrix.

Always refer to the current eMatrix Release Bulletins for any changes since the publication of this manual.

Conventions

Throughout this guide certain words appear in all capital letters, which indicates that you need to enter specific information for your environment. Below is a description of some of the abbreviations that are used in this manner.

MATRIXHOME: the directory that contains eMatrix applications.

OBJECT ID: The system-generated object ID number (OID) or Type Name Revision of the business object.

NAME: the specific name for an object



Part I:

General Functions

Introduction to MQL

MQL Defined.....	35
Three Modes: Interactive, Script, and Tcl.....	36
Script Mode Advantages.....	36
Using MQL.....	36
Command Line Editing	37
Tcl/Tk Mode	39
Tcl Versions	41
MQL and Tcl	42
Accessing MQL.....	43
MQL Command Options	44
-b Option	44
-c Option	45
-d Option	45
-k Option	45
-t Option	45
-v Option	45
-stdout:FILENAME	45
-stdin:FILENAME	45
-stderr:FILENAME	46
Statements and Scripts.....	47
MQL Statements	48
Entering (Writing) MQL Statements.....	48
Reviewing Statements	49
Important eMatrix Statements.....	50
Building an MQL Script	53
Running Scripts and Using Basic MQL Statements	53
Using Comments	54
Building an Initial eMatrix Database	54
Modifying an Existing Database	56
General Syntax.....	57
Add Statement.....	57
Copy Statement.....	57
Modify Statement.....	58
List Statement.....	58
List Admintype Statement.....	59
Print Statement.....	61
Delete Statement.....	62
Help Statement.....	62
Icon Clause.....	63

In Vault Clause	63
Evaluating Expressions.....	64
Business Object Collection Clause	64
Dump Clause	67
Recordseparator Clause	67
Examples	67
Tcl Format Select Output.....	68

MQL Defined

MQL is the Matrix Query Language. Similar to SQL, MQL consists of a set of statements that help the administrator set up and test an eMatrix database quickly and efficiently.

MQL is primarily a tool for building the eMatrix database. You can also use MQL to add information to the existing database, and extract information.

Three Modes: Interactive, Script, and Tcl

MQL acts an interpreter for eMatrix and can be used in one of three modes:

- Interactive Mode
- Script Mode
- Tool Command Language (Tcl) Mode

When you are using MQL in the interactive mode, you type in one MQL statement at a time. As each statement is entered, eMatrix processes it. This is similar to entering system commands at your terminal.

The interactive mode is useful if you have only a few commands to enter or want extensive freedom and flexibility when entering commands. However, interactive mode is very inefficient if you are building large databases or want to input large amounts of information. For this reason, eMatrix enables you to use MQL in a script mode.

When working in the script (or batch) mode, you use a text editor to build a set of MQL statements. These statements are contained in an external file, called a *script*, that can be sent to the eMatrix command interface. A script passes a batch of MQL statements to the eMatrix command interface for processing. The interface then reads the script, line by line, and processes the statements just as it would in the interactive mode.

Script Mode Advantages

Working in script mode has many advantages, particularly when you are first building a database. Some examples of advantages are:

- You have a written record of all your definitions and assignments. This enables you to review what you have done and make changes easily while you are testing the database. When you are first building the database, you may experiment with different definitions to see which one works best for your application. A written record saves time and aggravation since you do not have to print definitions when there is a question.
- You can use text editor features to duplicate and modify similar definitions and assignments. Rather than entering every statement, you can copy and modify only the values that must change. This enables you to build large and complicated databases quickly.
- Testing and debugging the database is simplified. MQL databases can be wiped clean so that you can resubmit and reprocess scripts. This means that you can maintain large sections of the MQL statements while easily changing other sections. Rather than entering statements to modify the database, you can simply edit the script. If you are dissatisfied with any portion of the built database, you can change that portion of the script without eliminating the work you did on other portions. (If you were working interactively, you would have to track what had and had not changed.) Since the script contains the entire database definition and its assignments, there is no question as to what was or was not entered.

Using MQL

Many MQL statements define structures and objects. When you are first setting up the eMatrix database, you will include these statements in an initial database building script or collection of scripts.

We recommend that you use MQL scripts as long as you are in the building process.

Later, you may decide to add information and files into eMatrix. The process of adding information may require additional MQL statements. Rather than entering them interactively, you can create a new script to handle the new modifications.

Using MQL interactively is most practical when you have only a few modifications to make or tests to perform.

After the database is built, you will most likely maintain it through the Business Modeler interface. This interface helps you see what you are changing.

When you work with the database interactively, you do not have a written record to help you retrace what you did. In the interactive mode, the MQL statements are performed immediately and not recorded. You are left with only the result of the statement. If you are inserting a large amount of information, it may be cumbersome to track all entries and modifications. Also, if you decide that it is better to start over again, it would mean losing everything that was done up to that point.

Command Line Editing

Command line editing is available in interactive MQL. MQL maintains a history list of commands and allow for editing of these commands using the arrow keys and some control characters.

To edit, use the arrow keys and move your cursor to the point where a change is required. Insert and delete characters and/or words, as needed.

Using Control Characters

All editing commands are control characters which are entered by holding the Ctrl or Esc key while typing another key, as listed in the following table. All editing commands operate from any place on the line, not just at the beginning of the line.

Command Line Editing Control Characters	
Ctrl-A	Move the cursor to the beginning of the line.
Ctrl-B	Move the cursor to the left (back) one column.
Esc-B	Move the cursor back one word.
Ctrl-D	Delete the character to the right of the cursor.
Ctrl-E	Move the cursor to the end of the line.
Ctrl-F	Move the cursor right (forward) one column.
Esc-F	Move the cursor forward one word.
Ctrl-H	Delete the character to the left of the cursor.
Ctrl-I	Jump to the next tab stop.
Ctrl-J	Return the current line.
Ctrl-K	Kill from the cursor to the end of the line (see Ctrl-Y).
Ctrl-L	Redisplay the current line.

Command Line Editing Control Characters	
Ctrl-M	Return the current line.
Ctrl-N	Fetch the next line from the history list.
Ctrl-O	Toggle the overwrite/insert mode, initially in insert mode.
Ctrl-P	Fetch the previous line from the history list.
Ctrl-R	<p>Begin a reverse incremental search through the history list. Each printing character typed adds to the search substring (which is empty initially). MQL finds and displays the first matching location. Typing Ctrl-R again marks the current starting location and begins a new search for the current substring.</p> <p>Type Ctrl-H or press the Del key to delete the last character from the search string. MQL restarts the search from the last starting location. Repeated Ctrl-H or Del characters, therefore, appear to unwind the search to the match nearest to the point where you last typed Ctrl-R or Ctrl-S (described below).</p> <p>Type Ctrl-H or press the Del key until the search string is empty to reset the start of the search to the beginning of the history list.</p> <p>Type Esc or any other editing character to accept the current match and terminate the search.</p>
Ctrl-S	Begin a forward incremental search through the history list. The behavior is like that of Ctrl-R but in the opposite direction through the history list.
Ctrl-T	Transpose the current and previous character.
Ctrl-U	Kill the entire line (see Ctrl-Y).
Ctrl-Y	<i>Yank</i> the previously killed text back at the current location.
Backspace	Delete the character left of the cursor.
Del	Delete the character right of the cursor.
Return	Return the current line.
Tab	Jump to the next tab stop.

The MQL Prompts

Interactive MQL has two prompts. The primary prompt is, by default:

MQL<%d>

where %d is replaced with the command number. The secondary prompt is, by default:

>

The secondary prompt is used when a new line has been entered without terminating the command.

You can change the primary and secondary prompts with the MQL command:

```
prompt [[PROMPT_1] [PROMPT_2]]
```

Without arguments, the prompts are reset to the defaults.

Tcl/Tk Mode

With Tcl (tool command language) embedded in MQL, common programming features such as variables, flow control, condition testing, and procedures are available for use with eMatrix. The Tk toolkit is also included. Tcl and Tk are widely available and documented. The information in this section is simply an overview of functionality. For more detailed procedures, consult the references listed in the section [For More Information](#).

Tcl Overview

Tcl is a universal scripting language that offers a *component approach* to application development. Its interpreter is a library of C procedures that have been incorporated into MQL. MQL/Tcl offers the following benefits to eMatrix users:

- **Rapid development**—Compared with toolkits where you program in C (such as the Motif toolkit), there is less to learn in order to use Tcl and less code to write. In addition, Tcl is an interpreted language with which you can generate and execute new scripts *on the fly* without recompiling or restarting the application. This enables you to test and fix problems rapidly.
- **Platform independence**—Tcl was designed to work in a heterogeneous environment. This means that a Tcl script that was developed under Windows can be executed on a UNIX platform. After a developer has completed design and testing, a program object can be created using the code providing eMatrix users with immediate access to the new application. This also solves the problem of a need to distribute new applications.
- **Integration support**—Because a Tcl application can include many different library packages, each of which provides a set of Tcl commands, an MQL/Tcl script can be used as a communication mechanism to allow different applications to work with eMatrix.
- **User convenience**—The `tcl` command has been added to MQL, enabling users to switch to Tcl mode. In addition, MQL commands can be executed while in Tcl mode by preceding the correct MQL syntax with `mql`.

Tcl Functionality

More specifically, Tcl enhances MQL by adding the following functionality:

- Simple text language with enhanced script capabilities such as condition clauses
- Library package for embedding other application programs
- Simple syntax (similar to sh, C, and Lisp):

Command	Output
<code>set a 47</code>	47

- Substitutions:

Command	Output
set b \$a	47
set b [expr \$a+10]	57

- Quoting:

Command	Output
set b "a is \$a"	47
set b {[expr \$a+10]}	[expr \$a+10]

- Variables, associative arrays, and lists
- C-like expressions
- Conditions, looping:

```
if "$x<<3" {
    puts "x is too small"
}
```

- Procedures
- Access to files, subprocesses
- Exception trapping

Using Tcl

To enter Tcl mode, enter the following in MQL:

```
tcl;
```

In Tcl mode, the default Tcl prompt is % and MQL will accept only Tcl commands. Native MQL commands are accessed from Tcl by prefixing the command with the keyword mql. For example:

```
% mql print context;
```

Otherwise, Tcl command syntax must be followed. It differs from MQL in several ways:

Tcl Command Syntax Differences	
Comments	If the first nonblank character is #, all the characters up to the next new line are treated as a comment and discarded. The following is not a comment in Tcl: set b 101; #this is not a comment
Command Separators	New lines are treated as command separators. They must be preceded by a backslash (\) to not be treated as such. A semi-colon (;) also is a command separator.
Commas	For MQL to properly parse commands with commas, the commas must be separated with blanks. For example: access read , write , lock
Environment Variables	Environment variables are accessible through the array, env. For example, to access the environment variable MATRIXHOME, use: env(MATRIXHOME)
History	Tcl has its own history mechanism.

Command line editing is not currently available.

The `Tcl` command, `exit`, returns you to native MQL. Like the MQL `quit` command, `exit` can be followed by an integer value which is passed and interpreted as the return code of the program.

Displaying Special Characters

When displaying special symbols in a Tk window, it may be necessary to specify the font to be used. Tcl/Tk 8.0 tries to map any specified fonts to existing fonts based on family, size, weight, etc. Depending on where you are trying to use a special symbol, you may be able to use the `-font arg`.

Tcl Versions

To determine what version of Tcl you are using, switch to `tcl` mode then use the `info tclversion` or `info patchlevel` commands:

```
MQL<1>tcl;
% info tclversion
8.0
```

Or

```
% info patchlevel
8.0
```

The Tk library must be loaded in order to retrieve the Tk version. On Windows, a command similar to the following should be used:

```
% load tk42
% info loaded
{tk42 Tk}
```

For More Information

For more information about Tcl and Tk, refer also to the following books:

Ousterhout, John K., *Tcl and the Tk Toolkit*. Addison-Wesley, April 1994, ISBN 0-201-63337-X.

Welch, Brent, *Practical Programming in Tcl and Tk*. Prentice Hall, May 1995, ISBN 0-13-182007-9.

World Wide Web documents for reference:

- USENET newsgroup: comp.lang.tcl

MQL and Tcl

When Tcl passes an MQL command to MQL, it first breaks the command into separate arguments. It breaks the command at every space, so you have to be careful to use double quotes ("") or braces ({}) to enclose any clause which is to be passed along to eMatrix as a single argument.

Parsing of Tcl statements containing backslashes within an MQL program object is different than when those same statements are run from MQL command prompt.

For example, create a text file with the command lines:

```
tcl;
set a "one\\two\\three"
puts "a contains $a"
puts "b will use slashes instead of backslashes"
regsub -all {\\"} $a {/} b
puts "now b contains $b"
```

Run this file from MQL (or alternatively type each line in MQL command window). The final line of output will be:

```
now b contains one/two/three
```

This is consistent with how the same lines of code behave in native Tcl.

However if you use this same code as an eMatrix MQL program object, to get the same output you need to use triple rather than double backslashes so your code becomes:

```
tcl;
set a "one\\\\two\\\\\\three"
puts "a contains $a"
puts "b will use slashes instead of backslashes"
regsub -all {\\\\\} $a {/} b
puts "now b contains $b"
```

Accessing MQL

There are several ways to access the MQL from your operating system.



For example, if you are working on a PC, select the icon.

Or

If you are working on a UNIX platform, enter the MQL command using the following syntax:

```
mql [options] [-] [file...]
```

Brackets [] indicate optional information. Do not include the brackets when you enter the command. **Braces { }** may also appear in a command syntax indicating that the items in the braces may appear one or more times.

mql	invokes the MQL interpreter
options	specifies one or more MQL command options
- (the hyphen)	specifies that entries come from standard input
file	the name of the script(s) to be processed

When your system processes the `mql` command, the eMatrix command interface starts up in one of two modes: interactive or script. The mode it uses is determined by the presence of the hyphen and file name(s):

- Interactive mode is used when the command includes the hyphen and/or does not include a file name(s). For example, to run MQL interactively, enter either:

```
mql -
```

Or:

```
mql
```

In both of these commands, no files are specified so the interpreter starts up in an interactive mode to await input from the assigned input device (usually your terminal).

- Script mode is used if one or more files are listed. Specifying a file indicates that eMatrix should process the script of MQL statements and then return control to the operating system.

For example, to process two script files, enter a command similar to the following.

```
mql TestDefinitions.mql TestObjects.mql
```

This command invokes the MQL interpreter, processes the MQL statements within the `TestDefinitions.mql` script, and then processes the statements within the `TestObjects.mql` script.

MQL Command Options

In addition to the hyphen and file name qualifiers, several MQL command options are available:

mql Command Options	Specifies that MQL...
-b FILENAME	Uses the bootstrap file FILENAME instead of matrix-r.
-c "command;command..."	Uses command;command... as the input script. Processes the MQL statements enclosed within the double quotes.
-d	Suppresses the MQL window but does not suppress title information.
-k	Does not abort on error. Continues on to the next MQL statement if an error is detected in an MQL statement. The -k option is ignored for interactive mode.
-q	Sets Quote on.
-t	Suppresses the printing of the opening eMatrix title and the MQL window.
-v	Works in verbose mode to print all generated messages.
-stdout:filename	Redirects MQL output to a file.
-stdin:filename	Reads MQL input from a file.
-stderr:filename	Redirects MQL errors to a file.

Each statement and command option is discussed in the following sections.

-b Option

By default, eMatrix reads matrix-r as the bootstrap file. The -b modifier indicates that an alternate bootstrap file should be used. The bootfile is expected to be in your MATRIXHOME directory. For example, suppose you have two databases that you frequently access, a test system (Mx_qtest) and a production system (Mx_production).

You could create two shortcuts on your Windows Start menu. The shortcut for the test database could be called QAmatrix and contain the following:

```
c:\matrix\bin\winnt\mql.exe -b Mx_qtest
```

The shortcut for the production database could be called PRDmatrix and contain the following:

```
c:\matrix\bin\winnt\mql.exe -b Mx_production
```

Generally, there is no need to duplicate or move eMatrix .ini files. However, if multiple files will be opened for view or edit from a database that is not using the standard matrix-r file, errors will occur.

-c Option

The `-c` option enables you to enter MQL statements from the system command line. This option specifies that eMatrix should process the MQL statements enclosed within the double quotes.

For example, the following command assigns new telephone and facsimile numbers to a defined user (Joe) from the system command line:

```
mql -c "modify person Joe phone 598-4354 FAX 598-4355;"
```

You can include more than one MQL statement in the command string. In the syntax `COMMAND ; {COMMAND ; }` you can include as many statements as you want, providing each statement ends with a semicolon and space and all the MQL statements are enclosed within the quotes.

-d Option

The `-d` option suppresses MQL window, but not the title information. See also [-t Option](#).

-k Option

An abort-on-error is the default when eMatrix reads MQL statements from a script or file. eMatrix aborts when an error is encountered and returns you to your operating system. However, you may not want a script to terminate when an error is found during processing. The `-k` option specifies that eMatrix should continue on to the next MQL statement if an error is detected in an MQL statement. In this case, an error message is printed when the error is encountered and the script continues to run.

-t Option

The `-t` option suppresses the printing of the opening MQL title. This title identifies the product name, copyright information, and version number. When you use MQL frequently, the `-t` option saves time since the information is not displayed. This also suppresses the display of the MQL window. See also [-d Option](#).

-v Option

The `-v` option specifies that eMatrix should work in *verbose* mode. In this mode, eMatrix will print all messages generated during startup and the processing of statements. If you do not need this level of detail, you still receive error messages and other important MQL messages without the `-v` option.

-stdout:FILENAME

By default, MQL output is displayed in the MQL window. On the PC, use the `-stdout` option to redirect the MQL command output to a file. FILENAME is the name of the file which will contain the MQL output.

eMatrix on UNIX uses the UNIX standard file descriptors. You can use the standard UNIX redirection techniques to redirect MQL output to files.

-stdin:FILENAME

On the PC, use the `-stdin` option to read MQL input from a file. FILENAME is the name of the file from which the data is read.

eMatrix on UNIX uses the UNIX standard file descriptors.

-stderr:FILENAME

By default, MQL error messages are displayed in the MQL window. On the PC, use the `-stderr` option to redirect MQL error messages to a file. FILENAME is the name of the file that will contain the error messages.

eMatrix on UNIX uses the UNIX standard file descriptors. You can use the standard UNIX redirection techniques to redirect MQL output to files.

Statements and Scripts

An MQL command is called a *statement*. All statements perform an action within eMatrix. For example, actions might define new structures within the database, manipulate the existing structures, or insert or modify data associated with the database structures.

An MQL *script* is an external file that contains MQL statements (see the example below). You can create this file using any text editor. After you insert all the desired MQL statements, you can batch process the statements by specifying the name of the script file in the `mql` command.

```
#####
# Script file: SampleScript.mql
# These MQL statements define a vault and create an object.
#####
verbose on;
#
# Set context to eMatrix System Administrator's and define a new vault.
#
set context dba;
add vault "Electrical Parts";
output "Electrical Parts vault has been added"
#
# Set context to that of person Steve in vault "Electrical Subassemblies"
#
set context vault "Electrical Subassemblies" user Steve;
#
# Add a business object of type Drawing with name C234 A3
# and revision 0 and check in the file drawing.cad
#
output "now adding a business object for Steve";
add businessobject Drawing "C234 A3" 0
  policy Drawing
    description "Drawing of motor for Vehicle V7";
checkin businessobject Drawing "C234 A3" 0 drawing.cad;
#
quit;
```

MQL Statements

An MQL statement consists of a keyword and, optionally, clauses and values. MQL statements follow their own set of rules and conventions, as in any other programming or system language. In the following sections, you will learn about the conventions used in this book for entering, as well as reviewing, displayed MQL statements.

Entering (Writing) MQL Statements

You must follow a fixed set of syntax rules in order for eMatrix to interpret the statement correctly. If you do not, eMatrix may simply wait until the required information is provided or display an error message.

When in interactive mode, each statement is prompted by:

mql<#>

The syntax rules are summarized in the following table. If you have a question about the interpretation of a statement, see the chapter on working with that MQL category (refer to the table of contents).

Writing and Entering eMatrix Statements	
Statements	All statements consist of words each separated by one or more spaces, tabs, or a single carriage return.
	All statements begin with a keyword. Most keywords can be abbreviated to three characters (or the least number that will make them unique).
	NAMES, VALUES, etc. must be enclosed within single or double quotes when they have embedded spaces, tabs, newlines, commas, semi-colons, or pound signs.
	All statements end with either a semicolon (;) or double carriage return.
	For example, this is the simplest MQL statement. It contains one keyword and ends with a semicolon: quit;

Writing and Entering eMatrix Statements	
Clauses	Statements may or may not contain clauses.
	All clauses begin with a keyword.
	All clauses are separated with a space, tab, or a carriage return.
	The following example is a statement with two clauses separated by both carriage returns and indented spaces. add attribute "Ship's Size" description "Ship size in metric tons" type integer;
Values	Multiple values are separated within clauses by a comma (,). Values may be single or, if the keyword accepts a list of values, they can be specified either separately or in a list. For example: attribute Size, Length <i>Or:</i> attribute Size attribute Length
Comments	All comments begin with a pound sign (#) and end with a carriage return. For example: # list users defined within the database
	All comments are ignored by eMatrix and are used for your information only.

In the chapters that follow, you will learn more about the keywords, clauses, and values that make up MQL statements.

Reviewing Statements

When statements are displayed on your screen, the statements are displayed in *diagrams* which obey a set of syntax rules. These rules are summarized in the following table:

Reviewing Displayed eMatrix Statements	
keywords	All keywords are shown in lowercase. These words are required by MQL to correctly process the statement or clause. Though they are displayed in all lowercase in the syntax diagram, you can enter them in eMatrix as a mixture of uppercase and lowercase characters.
	All keywords can be abbreviated providing they remain distinctive. For example, you can use bus for businessobject.
Values	<p>User-defined values are shown in uppercase. The words in the syntax diagram identify the type of value expected. When entering a value, it must obey these rules:</p> <ol style="list-style-type: none"> 1. You can enter values in uppercase or lowercase. However, values are case-sensitive. Once you enter the case, later references must match the case you used. For example, if you defined the value as Size, the following values would <i>not</i> match: SIZE, size, SIZE. 2. You must enclose a value in double (" ") or single (' ') quotes if it includes spaces, tabs, carriage returns (new lines), commas, semicolons (;), or pound signs (#). A space is included in the following examples, so the values are enclosed in quotes: "Project Size" 'Contract Terms' 3. If you need to use an apostrophe within a value, enclose the value in quotes. For example: "Project's Size" 4. If you need to use quotes within a value, enclose the value in single quotes. For example: 'Contract "A" Terms'
[option]	<p>Optional items are contained within square brackets []. You are not required to enter the clause or value if you do not need it.</p> <p>Do not include the brackets when you are entering an option.</p>
{option}	<p>All items contained within braces { } can appear zero or more times.</p> <p>Do not include the braces when you are entering an option.</p>
option1 option2 option3	All items shown stacked between vertical lines are options of which you must choose one.

Important eMatrix Statements

When you first use MQL, there are two important MQL statements you should know: Quit and Help.

Quit

The quit statement exits the eMatrix command interface and returns control to the operating system. To terminate your MQL session, simply enter:

```
quit;mql<#>
```

As you can see, this statement follows the MQL rules of syntax. The keyword `quit` is followed by a semicolon.

When writing the code for program objects, it is sometimes necessary for the program to return an integer value to the system that ran the program. For this reason, the `quit` statement can be followed by an integer, as follows:

```
quit [INTEGER];
```

The `INTEGER` value is passed and interpreted as the return code.

Help

The help statement is available for various MQL statement categories. If you do not know which category you want or you want to get a listing of the entire contents of the help file, enter:

```
help all;
```

The `help all` statement prints the entire help file on your output device.

Eventually, you will probably want help only on a selected MQL category:

association	attribute	businessobject
context	cue	env
export	filter	form
format	group	import
license	location	mail
monitor	person	policy
process	program	property
query	relationship	relationshiprule
report	role	rule
server	set	site
store	thread	tip
toolset	transaction	type
user	vault	view
wizard	workflow	

To get help on any of these categories, enter:

```
help MQL_CATEGORY;
```

`help` is the keyword.

`MQL_CATEGORY` is one of the categories listed above.

If you have a question about the interpretation of a statement, see the chapter on working with that MQL category.

When you enter this statement, eMatrix displays a list of all the statements associated with the category along with the valid syntax. For example, assume you entered the following statement:

```
help context;
```

This statement will cause the following *diagram* to display:

```
set context [ITEM {ITEM}];  
  where ITEM is:  
    | person PERSON_NAME |  
    | password [PASSWORD_VALUE] |  
    | vault [VAULT_NAME] |  
  
print context;
```

This information indicates that there are two statements associated with the context category: the `set context` and `print context` statements.

The first statement has three clauses associated with it: the `person`, `password`, and `vault` clauses. The square brackets ([]) on either side of `PASSWORD_VALUE` mean that it is optional. If a clause is used, you must obey the syntax of the clause. In the `person` clause, the syntax indicates that a value for `PERSON_NAME` is required. Words shown in all uppercase indicate user-defined values. (Refer to [Entering \(Writing\) MQL Statements](#) for a complete description of syntax rules.)

Building an MQL Script

Scripts are useful when you are performing many changes to the eMatrix database. This is certainly true in the initial building process and it can be true when you are adding a number of old files or blocks of users into an existing database. MQL scripts provide a written record that you can review. Using a text editor makes it easy to duplicate similar blocks of definitions or modifications.

Running Scripts and Using Basic MQL Statements

When you are building a script, there are several MQL statements that help run other scripts and let you monitor the processing of the script MQL statements.

MQL Statements to Use When Building a Script	
<code>output VALUE;</code>	<p>The Output statement enables you to print a message to your output device. This is useful within scripts to provide update messages and values while the script is processed.</p> <p>For example, if you are processing large blocks of statements, you may want to precede or end each block with an Output statement. This enables you to monitor the progress of the script.</p>
<code>password PASSWORD;</code>	<p>The Password statement enables you to change the current context password.</p> <p>The Password statement enables you to change your own password (which can also be done with the Context statement).</p>
<code>run FILE_NAME [continue];</code>	<p>The Run statement enables you to run a script file. This is useful if you are working in interactive mode and want to run a script.</p> <p>The continue keyword allows the script to run without stopping when an error occurs. This is essentially the same as running MQL with the -k option, but it is available at run time, making it usable by programs.</p>
<code>shell COMMAND;</code>	<p>The Shell statement enables you to execute a command in the operating system. For example, you can perform an action such as automatically sending an announcement to a manager after a set of changes are made.</p> <p>The Output statement (see above) sends a message to your output device; but, it cannot send a message to a different device. You can do so with the Shell statement.</p>

MQL Statements to Use When Building a Script	
verbose [on off];	<p>The Verbose statement enables you to increase or decrease the amount of message detail generated by the MQL interpreter as the script is processed. This is similar to the -v Option.</p> <p>When set to ON, more detail is provided. When set to OFF, only errors and important system messages are displayed.</p>
version;	The Version statement enables you to see which version of MQL you are using. For example, this is useful if you want a record of the version used to process a script.

Using Comments

Comments visually divide scripts into manageable sections and remind you of why structures and objects were defined and modified. Comments, which are preceded in the script by a pound sign (#), are ignored by MQL:

```
# script comments
```

Each comment line must begin with a # sign.

Building an Initial eMatrix Database

Building an eMatrix database usually involves writing two scripts. One script will contain all the definitions used by the database. The second script creates business objects, associates files with the objects, and defines relationships between objects.

By separating these two steps, it is easier to test the database. You cannot create business objects or manipulate them until the definitions are in place. By first creating and testing all the definitions, you can see if the database is ready to add objects. You may want to alter the definitions based on test objects and then use the altered definitions for processing the bulk of the object-related data.

Clearing the Database

When you are creating an initial database and you want to start over, the Clear All statement enables you to work with a clean slate. The Clear All statement clears all existing definitions, vaults, and objects from the database. It wipes out the entire content of the eMatrix database and leaves only the eMatrix shell. While this is useful in building new databases, it should NOT be used on an existing database that is in use.

Once this statement is processed, the database is destroyed and can only be restored from backups. To access the current database at any time in the future, you should make a backup before using this statement.

Only the person named “creator” can use the Clear All statement. (If “creator” no longer exists, you must redefine this person.)

```
clear all;
```

To assist you in building and testing new vaults, eMatrix provides the Clear Vault statement. This statement ensures that the vault does not contain any business objects. It can be run by a person with System Administrator privileges.

```
clear vault VAULT_NAME;
```

VAULT_NAME is the names of the vault(s) to be cleared.

When the Clear Vault statement is processed, eMatrix clears all business objects within the named vaults. The definitions associated with the vault and the eMatrix database remain intact. Only the business objects within the named vaults are affected.

The clear vault or clear all statements should NEVER be used while users are on-line.

Creating Definitions in a Specific Order

When creating the script of definitions for the initial database, the order in which you make the definitions is significant. Some statements are dependent on previous definitions. For example, you cannot assign a person to a group if that group does not already exist. For this reason, it is recommended that you write your definition statements in the following order:

Definition Order		
1	Roles Groups Persons Associations <i>Or:</i> Persons Groups Roles Associations	You can define roles, groups, and persons either way depending on your application. Since associations are combinations of groups and roles, they must be created after groups and roles are defined.
2	Attributes Types Relationships Formats Stores Policies	Order is important for this set of definitions. For example, types use attributes, so attributes must be defined before types.
3	Vaults	Vaults can be defined at any time although they are most commonly defined last.
4	Reports Forms	If reports or forms are required, they are usually defined after the database is built and tested. These definitions require workable values for reporting. If there are no values to report, it is difficult to define and test.

Processing the Initial Script

After your script is written, you can process it using the `mql` command, described in [Accessing MQL](#). As it is processed, watch for error messages and make note of any errors that need correction.

Once the script that creates the definitions is successfully processed, you can use either the interactive MQL interpreter or the Business Administrator account to check the definitions. From eMatrix Navigator or MQL, you can create objects, associate files, and otherwise *try out* the definitions you have created. If you are satisfied with the definitions, you are ready to write your second MQL script.

Writing the Second MQL Script

The second MQL script in this example creates business objects, manipulates them, and defines relationships. For more information on the types of information commonly found in this second script file, see [Business Objects](#) in Chapter 34.

Modifying an Existing Database

After you define the initial eMatrix database, you may want to modify it to add additional users and business objects. If the number of users or the amount of information is large, you should consider writing a MQL script. This enables you to use a text editor for editing convenience and provides a written record of all of the MQL statements that were processed.

Often when you are adding new business objects, you will define new vaults to contain them. This modular approach makes it easier to build and test. While some new definitions may be required, it is possible that you will use many of the existing definitions. This usually means that the bulk of the modification script will involve defining objects, manipulating them, and assigning relationships. These actions are done within the context of the vault in which the objects are placed.

To assist you in building and testing new vaults, eMatrix provides the `Clear Vault` statement. This statement ensures that the vault does not contain any business objects. See [Clearing the Database](#) for details.

When writing your script for modifying an existing database, remember to include comments, occasional output statements, and a `Quit` statement at the end (unless you'll want to use the interactive MQL interpreter after script completion).

General Syntax

The remainder of this chapter presents the syntax for statements that are common to many MQL features (*items* such as Vault, Person, Group, etc.).

Add Statement

The Add statement will create an instance of an item. The Add statement generally has many clauses and, in some cases, subclauses specific to each item. Syntax:

```
add ITEM ITEM_NAME [ADD_ITEM {ADD_ITEM}];
```

ITEM can be any of the following:

association	format	person	role	tip
attribute	group	policy	rule	toolset
businessobject*	index	process	server	type
command	inquiry	program	set*	vault
cue	location	query	site	view
filter	menu	relationship	store	wizard
form	page	report	table	workflow

* slightly different syntax depending on the item. Refer to the specific item chapter for details.

ITEM_NAME is a unique name for that kind of item.

ADD_ITEMS are applicable fields for each item with defined values. Refer to the specific item chapters for more information.

Copy Statement

The Copy statement will clone a definition, rename it, and make the specified modifications. Syntax:

```
copy ITEM SRC_NAME DST_NAME [MOD_ITEM {MOD_ITEM}];
```

ITEM can be any of the following:

association	format	person	role	tip
attribute	group	policy	rule	toolset
businessobject*	index	process	server	type
command	inquiry	program	set*	vault
cue	location	query	site	view
filter	menu	relationship	store	wizard
form	page	report	table	workflow

* slightly different syntax:

```
copy bus NAME to NEWNAME [Rev] [Vault]
```

SRC_NAME is the original (source) item name.

DST_NAME is the new (destination) item name.

MOD_ITEMS are applicable fields for each item with new values. Refer to the specific item chapters for more information.

Modify Statement

The Modify statement will make the specified changes to an existing definition.

```
modify ITEM NAME [MOD_ITEM {MOD_ITEM}];
```

ITEM can be any of the following

association	format	person	role	tip
attribute	group	policy	rule	toolset
businessobject*	index	process	server	type
command	inquiry	program	set*	vault
cue	location	query	site	view
filter	menu	relationship	store	wizard
form	page	report	table	workflow

* slightly different syntax depending on the item. Refer to the specific item chapter for details.

NAME is a unique name for that kind of item.

MOD_ITEMS are applicable fields for each item with new values. Refer to the specific item chapters for more information.

List Statement

The List statement is used to display all instances of an item that are currently defined. It is useful for confirming the existence or exact name of an item since eMatrix is case-sensitive.

```
list ITEM;
```

ITEM can be any of the following

association	format	person	role	tip
attribute	group	policy	rule	toolset
businessobject*	index	process	server	type
command	inquiry	program	set*	vault
cue	location	query	site	view
filter	menu	relationship	store	wizard

form	page	report	system	workflow
			table	toolset

*

List Admintype Statement

The List statement is used to display all instances of an administrative type that are currently defined. The List Admintype statement allows wildcards, lists and select clauses.

The full syntax of the List Admintype command is:

```
list ADMINTYPE [modified after DATE | NAME_PATTERN] [SELECT]
[DUMP [RECORDSEP]] [tcl] [output FILENAME];
```

ADMINTYPE can be any of the following

association	group	policy	rule	toolset
attribute	index	process	server	type
command	inquiry	program	set*	vault
cue	location	query	site	view
filter	menu	relationship	store	wizard
form	page	report	table	workflow
format	person	role	tip	

DATE must be in the format specified in .ini file. You cannot include a NAME_PATTERN when using modified after clause.

NAME_PATTERN can be a single name or a comma-delimited list of expressions that can include wildcard characters. You cannot include a modified after DATE clause when using NAME_PATTERN.

SELECT specifies a subset of the list contents.

DUMP allows you to insert formatting data into the printed information.

RECORD_SEP specifies a separator character for the output in the case of multiple objects.

tcl returns the results in Tcl list format.

FILENAME identifies a file where the print output is to be stored.

Each of these clauses is explained in detail in the sections that follow.

Name Pattern

NAME_PATTERN can be a comma-delimited list of expressions that can include the wildcard characters '*' or '?', where * matches any string of characters, and ? matches any single character. For example, if this command:

```
list person a*
```

produces this list: abadi,abami,adadi,adami,adams,apkarian,ata

then:

Command	Produces:
list person a?ami	adami,abami
list person ada?i	adami,adadi
list person a*i	adami,adadi,abami,abadi,at ai
list person ad*,a*n	adami,adams,apkarian,adadi

Select Clause

The Select clause of the List Admin type statement lets you specify a subset of the list content. For example, this command:

```
list attribute modified after DATE;
```

uses the modified selectable clause for the attribute admin type. The result would be a list of all attributes modified after the date specified. The DATE must follow the format specified in the initialization file.

For a list of all the selectable fields for each type of administrative object, see the *Select Expressions* appendix in the *eMatrix Navigator Guide*.

Dump Clause

You can specify a general output format with the Dump clause. The Dump clause specifies that you do not want to print the leading field names. You can also specify which character you want to use to separate the field names in the output:

```
dump ["SEPARATOR_STR"]
```

SEPARATOR_STR is a character or character string that you want to appear between the field values. It can be a tab, comma, semicolon, carriage return, etc. If you do not specify a separator string value, the default value of a comma is used. If tabs or carriage returns are used, they must be enclosed in double quotes ("").

When using the Dump clause, all the field values are printed on a single line unless a carriage return is the separator string.

For example, assume you entered the following statement:

```
list type Sales* select name description format dump;
```

The output might look like:

```
Sales Contacts West,west of Wakefield,HTML  
Sales Contacts East,Wakefield and east,HTML  
Sales Documents,Closing a Deal,PDF  
Separator strings can make the output more readable.
```

Record_Sep Clause

The Record_Sep clause of the List Admintype statement allows you to define which character or character string you want to appear between the dumped output of each object when the List command requests information about multiple objects.

```
recordseparator ["SEPARATOR_STR"]
```

SEPARATOR_STR is a character or character string that replaces the end-of-line (\n) ordinarily inserted at the end of each object's dumped output. It can be a tab, comma, semicolon, etc. If tabs are used, they must be enclosed in double quotes ("").

Tcl Clause

Use the Tcl clause after the dump clause, if used, and before the output clause to return the results in Tcl list format. This facilitates the parsing of output from MQL select commands within Tcl code since the built-in list handling features of Tcl are used directly on the results. For more information, see [Tcl Format Select Output](#).

Output Clause

The Output clause of the List Admintype statement provides the full file specification and path to be used for storing the output of the List statement. For example, to store information in an external file called sales.txt, you might write a statement similar to:

```
list type Sales* select name description format dump  
output c:\mydocs\sales.txt.
```

After DATE Clause

The After DATE clause of the List Admintype can be used to "query" based on modified dates. For example:

```
list attribute modified after DATE;
```

where DATE follows the format specified in the .ini file.

Print Statement

The Print statement prints the item definition to the screen allowing you to view it. When a Print statement is entered, MQL displays the various clauses that make up the definition.

```
print ITEM NAME;
```

ITEM can be any of the following:

association	format	person	role	tip
attribute	group	policy	rule	toolset
businessobject*	index	process	server	type
command	inquiry	program	set*	vault
cue	location	query	site	view

filter	menu	relationship	store	wizard
form	page	report	table	workflow

NAME is the specific instance of the item.

Delete Statement

The Delete statement enables you to delete an item when it is no longer needed in the eMatrix environment. Because deleting certain items can affect other elements of eMatrix, use it carefully. See the Delete section of each item chapter for a discussion of the impact of the deletion.

```
delete ITEM NAME;
```

ITEM can be any of the following:

association	group	policy	rule	toolset
attribute	index	process	server	type
businessobject*	inquiry	program	set*	user
command	location	query	site	vault
cue	mail	relationship	store	view
filter	menu	report	table	wizard
form	page	role	tip	workflow
format	person			

NAME is the specific instance of the item.

Help Statement

The Help statement is always available while you are in the MQL interactive mode. Help statements produce a summary of all statements applicable to the item with the appropriate syntax.

```
help ITEM;
```

ITEM can be any of the following:

all	form	page	role	toolset
association	format	person	rule	type
attribute	group	policy	server	user
businessobject*	index	process	set*	vault
command	inquiry	program	site	view
context	location	query	store	wizard
cue	mail	relationship	table	workflow
filter	menu	report	tip	

Icon Clause

The Icon clause often is available in a statement. It associates a special image with an item. For example, you may want to use a company logo for a Vault that contains objects related to doing business with that company. You could use a project logo for a project oriented Vault, or an object icon (such as a tax form) for the Vault containing those type of objects.

The GIF file needs to be accessible only during the definition using the Vault clause. Once an accessible file in the correct image type is used in the Vault clause, eMatrix will read the image and store it with the vault definition in the database. The physical icon files do not have to be accessible for every eMatrix session. (If the file is not accessible during definition, eMatrix will be unable to display the image.)

To write an Icon clause, you need the full directory path to the icon you are assigning. For example, the following statement assigns a calendar icon to the 1999 Records Vault:

```
add vault "1999 Records"  
icon $MATRIXHOME/pixmaps/app/calendar1999.gif;
```

You can set eMatrix Navigator to view images with the View menu options or Session>Preferences options.

The Icon clause of a definition statement is optional unless there is a need to distinguish icons. If you do not specify an icon, a default is used (such as `type.gif` for type definitions). When the default icon is desired, it should not be explicitly defined since doing so would affect performance. For example, if the default for Type A is used and Type B specifies the use of `type.gif`, two copies of the same file (`type.gif`) are read and processed each time these types are displayed. If only the default is used, the same information is processed once and displayed for all type icons.

Icons can be checked out of the objects that contain them. For syntax, see [Retrieving the Image](#) in Chapter 34 and the *Icon* sections of the [Working With Users](#) in Chapter 11.

In Vault Clause

When specifying business objects in MQL statements, the `in vault` clause is generally available. This clause improves the performance of such transactions as modify, delete, and connect, since it is no longer necessary to search all vaults to find the object to act upon. This is particularly true in a distributed or loosely-coupled database.

Evaluating Expressions

The `evaluate expression` command makes it possible to evaluate an expression without saving it in a Table. It also allows evaluation of a statistical expression against any collection of business objects that can be defined through various operations of combining, intersecting, or subtracting the collection of business objects from one Set, Query, temp query, or expand statement with/from another.

To evaluate an expression, use the Evaluate Expression statement:

```
evaluate expression EXPRESSION {EXPRESSION} on ON_ITEM {on ON_ITEM} [dump  
"SEPARATOR_STR"] [recordseparator "SEPARATOR_STR"];
```

`EXPRESSION` is an expression.

`ON_ITEM` can be any of the following:

`relationship ID`

`businessobject TYPE NAME REV`

`BUS_OBJ_COLLECTION_SPEC [{and | or | less} BUS_OBJ_COLLECTION_SPEC] ...`

Expressions can be evaluated on relationships. For information on how to find relationship IDs, refer to [Connection IDs](#) in Chapter 14.

Expressions can also be evaluated on a specified business object.

Expressions can also be evaluated on a collection of business objects, as explained below.

The dump and recordseparator clauses can occur anywhere in the command, but the EXPRESSIONs must be given before any ON_ITEMS.

This command outputs the result of evaluating each expression for each `ON_ITEM` one after the other.

- The dump separator character (a comma, by default) separates each value for a single `ON_ITEM`.
- The recordseparator character (a new line, by default) separates the results of one `ON_ITEM` from the next.

Business Object Collection Clause

Adding the concept of a `BUS_OBJ_COLLECTION_SPEC` extends the functionality of expressions and provides enormous flexibility. Any of the items listed below can be used in combinations covering multiple levels of complexity using the binary operators `and`, `or` and `less`.

Note that '+' and '-' cannot be used for the words 'or' and 'less.'

Any number of parentheses can be added around a `BUS_OBJ_COLLECTION_SPEC` or some part of one. `BUS_OBJ_COLLECTION_SPEC` is defined recursively as one of:

`set NAME`

`query NAME`

`temp set OBJECTID [{,OBJECTID}]`

```
temp query [TEMP_QUERY_ITEM {TEMP_QUERY_ITEM }]
```

```
expand [EXPAND_ITEM {EXPAND_ITEM}]
```

```
[ ( { ( } ]BUS_OBJ_COLLECTION_SPEC [ ){ )} ]
```

TEMP_QUERY_ITEM is one of:

```
businessobject OBJECTID
```

```
owner NAME
```

```
vault NAME
```

```
[ ! | not ]expandtype
```

```
where WHERE_CLAUSE
```

```
limit NUMBER
```

OBJECTID is the OID or Type Name Revision of the business object. It can also include the in VAULTNAME clause, to narrow down the search.

The values of TEMP_QUERY_ITEM have the same meaning in this context as they have for the temp query command, as described in [Defining a Saved Query](#) in Chapter 38.

EXPAND_ITEM is one of:

```
from
```

```
to
```

```
type PATTERN { ,PATTERN}
```

```
relationship PATTERN { ,PATTERN}
```

```
select businessobject
```

```
select relationship
```

```
where WHERE_CLAUSE
```

```
activefilters
```

```
reversefilters
```

```
filter PATTERN
```

```
view NAME
```

```
recurse
```

```
recurse to N
```

```
recurse to all
```

The values of EXPAND_ITEM have the same meaning in this context as they have for the expand bus command, as described in [Displaying and Searching Business Object Connections](#) in Chapter 35. (In the case of “select,” it indicates whether a subsequent ‘where clause’ applies to business objects or to relationships. In expand bus it has an additional meaning that is not applicable here.)

For example, the output from evaluating two expressions, E1 and E2, on two sets, A and B, with values V1A, V2A, V1B, and V2B would look like the following (using default separators):

V1A,V2A

V1B,V2B

Business Object Collection Specs can be linked with binary operators, which follow simple, intuitive rules:

- and—an object is in both collections
- or—an object is in one or the other collection
- less—an object is in the result if it is in the left-hand collection but not the right-hand one.

If there is more than one binary operator in a BUS_OBJECT_COLLECTION_SPEC, parentheses must be used to disambiguate the clause. (There is no implied order of operations.) You must include a space before and after each parenthesis. In addition, the number of left and right parentheses must match each other. For example:

```
( set A + set B ) - set C
```

would probably evaluate differently than:

```
set A + ( set B - set C )
```

Examples

This list of examples shows the power of the BUS_OBJECT_COLLECTION_SPEC concept.

The following statement returns the number of objects in the set Projects:

```
eval expression "count TRUE" on set Projects;
```

The same thing could be written as follows (note the spaces around the parentheses):

```
eval expression "count TRUE" on ( set Projects );
```

The following statement returns the number of objects returned by a query named “ask1” that are not Project 1029 0:

```
eval expression "count TRUE" on query ask 1 less temp set Project 1029 0;
```

The following statement returns the number of Jim’s open Projects that originated before the beginning of this year:

```
eval expression "count TRUE" on temp query bus Project * * owner Jim where "originated < '1/1/99' and current == open";
```

The following statement returns the number of open, new Projects that do not belong to Jim:

```
eval expression "count TRUE" on ( set openprojects AND set newprojects ) less temp query bus Project * * owner Jim;
```

The following statement returns the number of Projects that are either owned by Jim or included in the set “Q4 Projects”;

```
eval expression "count TRUE" on ( temp query bus Projects * * owner Jim ) OR set "Q4 Projects";
```

Dump Clause

You can specify a general output format for listing the information for output. The syntax is:

```
dump "SEPARATOR_STR"
```

SEPARATOR_STR is a character or character string (a comma, by default) that separates each value for a single ON_ITEM.

Recordseparator Clause

The Recordseparator clause of the Evaluate Expression statement allows you to define which character or character string you want to appear between the output of each ON_ITEM.

```
recordseparator "SEPARATOR_STR"
```

SEPARATOR_STR is a character or character string (a new line, by default) that separates the results of one ON_ITEM from the next.

Examples

The following example combines several of the capabilities available for expressions. It shows how to obtain the average and total number of days spent to move a feature from Open to Test for v7.0 and v7.1 through one MQL command. We divide by (3600 * 24) (the number of seconds in a day) because we want the number of days and the difference of two dates is given in seconds

```
evaluate expression "(average (state[Test].actual - state[Open].actual) )/ (3600 * 24)"  
    "( sum (state[Test].actual - state[Open].actual) ) / (3600 * 24)"  
on expand Product XYZ v7.0 to relationship Committed, Candidate, Proposed type Feature  
    where "current == Test or current == Closed"  
on expand Product XYZ v7.1 to relationship Committed, Candidate, Proposed type Feature  
    where "current == Test or current == Closed"  
dump " | ";
```

The next example calculates the number of Features connected to v7.2 that have been promoted to Test in the past week (and have not been demoted).

```
evaluate expression "count (( MX_CURRENT_TIME - state[Test].actual ) / (3600 * 24) < 7)"  
on expand XYZ 7.2 to relationship Committed, Candidate, Proposed type Feature  
    where "current == Test or current == Closed"  
dump " | ";
```

Tcl Format Select Output

MQL `print`, `list`, `expand`, and `temp query` commands that include selects accept an optional `tcl` clause to signal that results should be returned in Tcl list format. This facilitates the parsing of output from MQL select commands within Tcl code since the built-in list handling features of Tcl are used directly on the results.

Additionally, logic has been added that guarantees an entry in the results list for every given *legal* select item. This “padding” ensures that when the Tcl clause is included, the results list can be parsed properly. If illegal select items are given, there are no corresponding entries in the results list. Commands that don’t include the Tcl clause output as before; that is without this “padding.”

Since the Tcl list structure is already curly brace delimited, any given separator characters (specified with the `dump` or `recordseparator` clauses) are ignored when generating Tcl output. However, for readability, newlines are used as record separators between objects. For instance, in an `expand bus` command that includes selects and the Tcl clause, all selected items for one object are included on one line, and a new line is started for each connected object. Newlines are considered simple whitespace by Tcl, and so cause no parsing problems.

To avoid issues with Tcl list operations, the three special characters ‘{’, ‘}’, and ‘\’ are escaped by the ‘\’ character wherever they occur in text appearing between curly braces.

MQL select tcl clause

The following four MQL select commands may optionally use the “tcl” clause.

- `print`
- `list`
- `expand`
- `temporary query`

This clause must follow the `dump` clause, if used, and precede the `output` clause. In `temp` queries, it must also follow any `recordseparator` clause used. For example:

```
print bus TYPE NAME REVISION [SELECT] [dump [RECORDSEP]] [tcl] [output FILENAME];
print set SETNAME [SELECT] [dump [RECORDSEP]] [tcl] [output FILENAME];
list ADMINTYPE NAME_PATTERN [SELECT] [dump [RECORDSEP]] [tcl] [output FILENAME];
expand bus TYPE NAME REVISION [SELECT] [dump RECORDSEP] [tcl] [output FILENAME];
expand set SETNAME [SELECT] [dump RECORDSEP] [tcl] [output FILENAME];
temp query TYPE_PATTERN NAME_PATTERN REV_PATTERN QUERY_EXP [SELECT] [dump [RECORDSEP]]
[recordseparator SEPARATOR_STR] [tcl] [output FILENAME];
```

Notice that when expanding both business objects and sets and including the `dump` clause, a `RECORDSEP` is required. However, when the `Tcl` clause is also included, the separator is ignored. For justification, refer to Output Example number 11.

Output Format

The output from each Tcl select command is consistent with the expected MQL output, except that it uses Tcl list format. The results for each object are wrapped in a list, even

when one object is returned, as is the case with the print command. Within each object record is a header section followed by a sequence of select result items, like so:

```
{ object1-header-data      select-result-items }  
{ object2-header-data      select-result-items }  
:  
:
```

Header Data

The format of the header data depends on the actual command.

- Print, List, and Temporary Query Commands
 - business object:{business object} {TYPE} {NAME} {REV}*
 - connection object:{connection} {ID}*
 - administration:{TYPE} {NAME}*
 - set: no set header - each member uses:{member business object} {TYPE} {NAME} {REV}*
- Expand Command
 - {LEVEL} {REL_NAME} {FROM/TO} {TYPE} {NAME} {REV}*

Select Results Data

Each item in the select-result-items is in the format:

```
{ {select-item} results-list }
```

where each item in the results-list is:

```
{ {name} {value} }
```

or the results-list is simply {} if an empty result.

Using the dump clause

When the dump clause is specified, most of the header is dropped, the **{select-item}** is dropped, and just **{value}** is given for each item in the results-list.

Some commands (like “expand”) require a RECORDSEP to be included after the “dump” keyword. When used with the Tcl clause, this character is still required, but it is ignored.

Output Examples

1. Simple print business object with Tcl clause.

```
MQL<2>print bus Assembly SA-300356 0 select id owner  
locker tcl;  
{business object} {Assembly} {SA-300356} {0} {{id} {{id}  
{20083.46775.20193.6352}}} {{owner} {{owner} {Joe  
Product-Manager}}} {{locker} {{locker} {bucknam}}}
```

It may appear that there is some redundancy in the output in the select-result-items section. The first occurrence of an item (for example owner, above) matches the given select item and is followed by name/value pairs that resulted from the select item. Often the name field is redundant. However, when the select item results in many returned items (for example attribute.value or from.id) the name field gives added details. Refer to example 3 below.

2. Same print bus command including the dump clause:

```
MQL<3>print bus Assembly SA-300356 0 select id owner locker dump tcl;
```

```
{ {20083.46775.20193.6352} } { {Joe Product-Manager} }  
{ {bucknam} }
```

3. Print a business object with a select that returns multiple items:

```
MQL<4>print bus Assembly SA-300356 0 select from.id tcl;  
{business object} {Assembly} {SA-300356} {0} {{from.id}  
{{from[Documentation].id} {20083.46775.30402.24363}}}  
{{from[Analysis].id} {20083.46775.30631.62767}} {{  
from[Analysis].id} {20083.46775.30631.41948}}  
{{from[Plan].id} {20083.46775.30632.60059}}  
{{from[Required Tools].id} {20083.46775.30639.24886}}  
{{from[BOM-As Designed].id} {20083.46775.30663.9546}}  
{{from[BOM-As Designed].id} {20083.46775.6577.56182}}
```

4. Same print of multiple items with the dump clause, eliminating headers:

```
MQL<5>print bus Assembly SA-300356 0 select from.id dump tcl;  
{ {20083.46775.30402.24363} {20083.46775.30631.62767}  
{20083.46775.30631.41948} {20083.46775.30632.60059}  
{20083.46775.30639.24886} {20083.46775.30663.9546}  
{20083.46775.6577.56182}}
```

5. Temporary query with simple select in Tcl list output (with newline between object records):

```
MQL<6>temp query bus Assembly SA* 0 limit 5 select id  
locker tcl;  
{businessobject} {Assembly} {SA-300.125} {0} {{id} {{id}  
{20083.46775.31292.44899}}} {{locker} {{locker} {}}}  
{businessobject} {Assembly} {SA-300127} {0} {{id} {{id}  
{20083.46775.20133.58276}}} {{locker} {{locker}  
{bucknam}}}  
{businessobject} {Assembly} {SA-300195} {0} {{id} {{id}  
{20083.46775.20117.54372}}} {{locker} {{locker}  
{bucknam}}}  
{businessobject} {Assembly} {SA-300315} {0} {{id} {{id}  
{20083.46775.20173.48444}}} {{locker} {{locker} {}}}  
{businessobject} {Assembly} {SA-300356} {0} {{id} {{id}  
{20083.46775.20193.6352}}} {{locker} {{locker}  
{bucknam}}}
```

6. Same temp query with dump clause:

```
MQL<7>temp query bus Assembly SA* 0 limit 5 select id  
locker dump tcl;  
{Assembly} {SA-300.125} {0} {{20083.46775.31292.44899}}  
{{}}  
{Assembly} {SA-300127} {0} {{20083.46775.20133.58276}}  
{{bucknam}}  
{Assembly} {SA-300195} {0} {{20083.46775.20117.54372}}  
{{bucknam}}  
{Assembly} {SA-300315} {0} {{20083.46775.20173.48444}}  
{{}}
```

```
{Assembly} {SA-300356} {0} {{20083.46775.20193.6352}}
```

```
{bucknam}}
```

7. Temp query selecting attributes and their values:

```
MQL<8>temp query bus * * * limit 1 select current revision  
attribute.value tcl;
```

```
{businessobject} {NC Program} {GH02456} {A} {{current}}
```

```
 {{current} {Planned}}} {{revision} {{revision} {A}}}
```

```
 {{attribute.value} {{attribute[MachineType].value}}
```

```
 {Machine Center}} {{attribute[Process Type].value}}
```

```
 {Unknown}} {{attribute[Written By].value} {}}
```

```
 {{attribute[File Suffix].value} {.TAP}}}}
```

8. Same temp query with dump clause:

```
MQL<9>temp query bus * * * limit 1 select current revision  
attribute.value dump tcl;
```

```
{NC Program} {GH02456} {A} {{Planned}} {{A}} {{Machine
```

```
Center} {Unknown} {}{.TAP}}}
```

9. Expand set with newlines for each object:

```
MQL<10>expand set "A1 - To Do" limit 5 select bus id tcl;
```

```
{1} {Documentation} {to} {Drawing} {SA-300356} {A} {{id}}
```

```
 {{id} {20083.46775.30402.28967}}}
```

```
{1} {Analysis} {to} {Cost Analysis} {DA-3001356-1} {A}
```

```
 {{id} {{id} {20083.46775.65320.27011}}}
```

```
{1} {Analysis} {to} {Design Analysis} {DA-3001356-1} {C}
```

```
 {{id} {{id} {20083.46775.26552.19182}}}
```

```
{1} {Plan} {to} {Assembly Process Plan} {SA-300356} {0}
```

```
 {{id} {{id} {20083.46775.62627.22040}}}
```

```
{1} {Required Tools} {to} {Setup Instructions}
```

```
 {SA-300356} {A} {{id} {{id} {20083.46775.17589.27146}}}}
```

10. Same expand set with dump clause:

```
MQL<11>expand set "A1 - To Do" limit 5 select bus id dump  
: tcl;
```

```
{1} {Documentation} {to} {Drawing} {SA-300356} {A}
```

```
 {{20083.46775.30402.28967}}
```

```
{1} {Analysis} {to} {Cost Analysis} {DA-3001356-1} {A}
```

```
 {{20083.46775.65320.27011}}
```

```
{1} {Analysis} {to} {Design Analysis} {DA-3001356-1} {C}
```

```
 {{20083.46775.26552.19182}}
```

```
{1} {Plan} {to} {Assembly Process Plan} {SA-300356} {0}
```

```
 {{20083.46775.62627.22040}}
```

```
{1} {Required Tools} {to} {Setup Instructions}
```

```
 {SA-300356} {A} {{20083.46775.17589.27146}}
```

11. Expand bus with dump clause but no RECORDSEP:

MQL<12>expand bus Vehicle M60000 0 from recurse to 1 select rel id dump tcl;

```
1tclDOCUMENTATIONtcltotclManualtclM60001tcl0tcl974.54590.
```

```
24670.436
```

```
1tclDesigned Part QuantitytcltotclFront  
AxletclM66000tcl0tcl974.54590.42602.58431
```

12. A typical Tcl program might look like:

```
set output [mql expand set "A1 - To Do" select bus id dump  
: tcl]  
set count [llength $output]  
foreach row $output {  
    set level [lindex $row 0]  
    set relation [lindex $row 1]  
    set tofrom [lindex $row 2]  
    set busobj [lrange $row 3 5]  
    set busid [join [lindex $row 6]]  
    puts "\[level: $level\] relationship\[{$relation}\]  
$tofrom $busid"  
}
```

With results similar to:

```
[level: 1] relationship[Documentation] to  
20083.46775.30402.28967  
[level: 1] relationship[Analysis] to  
20083.46775.65320.27011  
[level: 1] relationship[Analysis] to  
20083.46775.26552.19182  
[level: 1] relationship[Plan] to 20083.46775.62627.22040  
[level: 1] relationship[Required Tools] to  
20083.46775.17589.27146  
[level: 1] relationship[BOM-As Designed] to  
20083.46775.30337.38798  
[level: 1] relationship[BOM-As Designed] to  
20083.46775.29481.42882  
[level: 1] relationship[Product Spec to BOM] from  
20083.46775.32142.28396  
:
```

2

Working With Transactions

Implicit and Explicit Transactions.....	74
Implicit Transactions	74
Explicit Transaction Control	74
Working With Threads.....	77
Start Thread Command	77
Resume Thread Command	77
Print Thread Command	77
Kill Thread Command	77

Implicit and Explicit Transactions

A *transaction* involves accessing the database or producing a change in the database. All MQL statements involve transactions.

Implicit Transactions

When you enter an MQL statement, the transaction is started and, if the statement is valid, the transaction is *committed* (completed). For example, assume you enter the following MQL statement:

```
add person Debbie;
```

As soon as you enter the statement, eMatrix starts the transaction to define a person named Debbie. If that person is already defined, the statement is invalid and the transaction aborts—the statement is not processed and the database remains unchanged. If the person is not already defined, eMatrix is committed to (the completion of) the transaction of adding a new person. Once a transaction is committed, the statement is fully processed and it cannot be undone. In this case, Debbie would be added to the database.

Ordinarily, starting, committing, and aborting transactions is handled *implicitly* with every MQL statement. Each statement has two implied boundaries: the starting keyword at the beginning of the statement and the semicolon or double carriage return at the end. When you enter an MQL statement, a transaction is implicitly started at the keyword and is either committed or aborted depending on whether the content of the statement is valid.

This implicit transaction control can also be performed explicitly.

Explicit Transaction Control

Several MQL statements enable you to explicitly start, abort, and commit transactions:

```
abort transaction [NAME];  
commit transaction;  
print transaction;  
start transaction [read];  
set transaction wait|nowait|savepoint [NAME];
```

These statements enable you to extend the transaction boundaries to include more than one MQL statement. If you are starting a transaction, use the `read` option if you are only reading. Without any argument, the `start transaction` command allows reading and modifying the database.

Extending Transaction Boundaries to Include Several Statements

Including several MQL statements within a single set of transaction boundaries enables you to tie the success of one statement to the success of some or all of the other MQL statements.

It will appear that all valid MQL statements are performed; but, they are not permanent until they are committed. You should not quit until you terminate the transaction or you

will lose all your changes. Also, the longer you wait before committing changes, the more likely you are to encounter an error message, particularly if you are entering the statements interactively when typing errors are common.

Let's look at an example in which you want to create a set of business objects and then associate a collection of files with those objects. You might successfully create the objects and then discover that you cannot place the files in them. With normal script or interactive MQL processing, the objects are created even though the checkin fails. By altering the transaction boundaries, you can tie the successful processing of the files to the creation of the business objects to contain them. For example:

```
start transaction update;
add businessobject Drawing "C234 A3" 0
    policy Drawing
        description "Drawing of motor for Vehicle V7";
checkin businessobject Drawing "C234 A3" 0 V7-234.MTR;
add businessobject Drawing "C234a A3" 0
    policy Drawing
        description "Drawing of alt. motor for Vehicle V7";
checkin businessobject Drawing "C234a A3" 0 V7-234a.MTR;
commit transaction;
```

This transaction is started and the current state of the database is saved. The `add businessobject` statements create business objects, and the `checkin businessobject` statements add the files.

When the `commit transaction` statement is processed, MQL examines all the statements that it processed since the `start transaction` statement. If no error messages are detected, all changes made by the statements are permanently made to the database.

If ANY errors are detected, NONE of the changes are committed. Instead, eMatrix returns the database to the recorded state it was in at the start of the transaction. This essentially gives you an *all or nothing* situation in the processing of MQL statements. However, savepoints can be set in the transaction to be used in conjunction with the `abort transaction` statement, which can save some of the work already done.

The `set transaction savepoint [NAME]` statement allows you to set a point within a transaction that will be used for any required rollbacks. If you specify the NAME in an `abort transaction` command, the transaction is rolled back to the state where the savepoint was created. If no name is specified, the entire transaction is rolled back. Of course the transaction must still be committed before the statements between the `start transaction` and the savepoint are processed.

Since DB2 does not support nested transaction savepoints, nested savepoints cannot be used in MQL transactions on DB2. An attempt to nest savepoints will result in a DB2 error stating that nested savepoints are unsupported.

The `wait` and `nowait` arguments are used to tell the system if you want to queue up for locked objects or not. When `nowait` is used, an error is generated if a requested object is in use. The default is to `wait`; the `wait` keyword is a toggle.

The Oracle SQL `nowait` function is not supported in DB2. Therefore in DB2, the `nowait` argument is ignored.

One advantage to using the transaction commands to process statement blocks involves the processing time. Statements processed within extended transaction boundaries are processed more quickly than the normal statement transactions.

If you choose to process blocks of MQL statements within explicit transaction boundaries, you need to carefully consider the size and scope of the statements within that transaction. When you access anything with an MQL statement, that resource is locked until the transaction is completed. Therefore, the more statements you include within the transaction boundaries, the more resources that will be locked. Since a locked resource cannot be accessed by any other user, this can create problems if the resource is locked at a time when heavy use is normal. The use of large transactions may also require you to adjust the size of your Oracle rollback segments. See your Oracle documentation for more details.

Be careful that the blocks are not too large.

You should immediately attend to an explicit transaction that either has errored or is awaiting a `commit` or `abort` command. Many database resources other than objects are also locked with the pending transactions. eMatrix users will begin to experience lock timeouts as they attempt typical database operations.

Working With Threads

MQL has a notion of a “thread,” which is a means by which you can run multiple MQL sessions inside a single process. Each thread is like a separate MQL session, except that all threads run within the same MQL process. This concept was introduced into MQL to support the MSM idea of multiple “deferred commit” windows.

Start Thread Command

By default, when you enter MQL, you are in thread number 1. You can start a new thread by running the `start thread` command. The command returns the ID (an integer) of the new thread. You are now in the new thread.

```
start thread;
```

If, in the previous thread, you had started a transaction and made a change to the database but had not committed it yet, the change will not be visible in the new thread. This behavior is the same as you would have if you had multiple MQL processes running and in one of them a transaction had been started and a change made.

Resume Thread Command

You can switch back to an existing thread using the `resume thread` command. Indicate the ID of the thread you want to resume. For example, if you are in the third thread and want to return to the first thread, use:

```
resume thread 1;
```

Print Thread Command

You can find out the current thread by running `print thread`. The ID number of the current thread is returned.

```
print thread;
```

Kill Thread Command

A thread other than the current one can be killed (like killing an MQL process) by using the `kill thread` command. Indicate the number of the the thread that you want to terminate.

```
kill thread 2;
```


Part II:

System Administrator Functions

3

Maintaining the System

System Administrator Responsibilities	82
Controlling System-wide Settings	83
Updating Sets With Change Vault	83
Database Constraints	84
System Decimal Symbol.....	85
Allowing Empty Strings for Object Names	85
Setting History Logging for the System	86
Privileged Business Administrators	86
System tidy	86
List Statement.....	87
Print Statement.....	87
Validating the eMatrix Database.....	88
Using an Oracle Database.....	89
Using a DB2 Database	89
Maintaining and Monitoring Clients.....	90
Viewing User Session Information.....	90
Error Log and Disk Space.....	90
Access Log	90
Monitoring Servers	95
Enabling Tracing.....	95
Enabling trace tools via MQL.....	97
ADK interfaces.....	98
Output.....	100
Tracing and Debugging	101
Examples	102
ADK Session Monitoring	105
Developing a Backup Strategy	110

System Administrator Responsibilities

The duties of the eMatrix System Administrator are to maintain and troubleshoot the system. The System Administrator should also be the on-site point of contact for all software updates, revisions, and customer support requests. Specific duties include:

- [*Controlling System-wide Settings*](#)
- [*Validating the eMatrix Database*](#)
- [*Maintaining and Monitoring Clients*](#)
- [*Monitoring Servers*](#)
- [*Developing a Backup Strategy*](#)

Other responsibilities, covered in other chapters include:

- [*Working With Vaults*](#) in Chapter 4
- [*Working With Stores*](#) in Chapter 5
- [*Replicating Captured Stores*](#) in Chapter 6
- [*Distributing the Database*](#) in Chapter 7
- [*Working With Export/Import*](#) in Chapter 9

Controlling System-wide Settings

Password requirements can be set for the entire system, as described in [System-Wide Password Settings](#) in Chapter 11. In addition, eMatrix System Administrators can control certain other settings that affect the system as a whole using the set system command:

```
set system SYSTEM_SETTING;
```

Where SYSTEM_SETTING is:

changevault update set on off
constraint index [indexspace SPACE] normal none
decimal . ,
emptyname on off
history on off
privilegedbusinessadmin on off
tidy on off

Updating Sets With Change Vault

When an object's vault is changed, by default the following occurs behind the scenes:

- The original business object is cloned in the new vault with all business object data except the related set data
- The original business object is deleted from the “old” vault.

When a business object is deleted, it also gets removed from any sets to which it belongs. This includes both user-defined sets and sets defined internally. IconMail messages and the objects they contain are organized as members of an internal set. So when the object's vault is changed, it is not only removed from its sets, but it is also removed from all IconMail messages that include it. In many cases the messages alone, without the objects, are meaningless. To address this issue, the following functionality can optionally be added to the change vault command:

- Add the object clone from the new vault to all IconMail messages and user sets in which the original object was included.

Since this additional functionality may affect the performance of the change vault operation if the object belongs to many sets and/or IconMails, it is not part of the function

by default, but business administrators can execute the following MQL command to enable/disable this functionality for system-wide use:

```
set system changevault update set | on | off | ;
```

For example, to turn the command on for all users, use:

```
set system changevault update set on;
```

Once this command has been run, when users change an object's vault via any application (that is, eMatrix desktop or Web applications, MatrixOne's Value Chain Portfolio applications, or custom ADK applications), all IconMails that reference the object are fixed, as well as user's sets. Refer also to [Changing an object's vault](#) in Chapter 34 for more information.

Database Constraints

Some versions of Oracle have a bug that limits performance and concurrency when an operation uses a column that has a foreign key constraint and that column is also not indexed. Such columns can cause deadlocks and performance problems. For systems that use foreign keys extensively, this leads to a trade off that must be made between performance, storage, and use of foreign keys. You can use the set system constraint command to tailor the eMatrix schema for one of three possible modes of operation::

```
set system constraint |none| ;  
|index [indexspace SPACE]|  
|normal|
```

- Normal

The normal system setting results in a schema that has foreign keys that are not indexed; that is, foreign key constraints are configured as they have been in pre-version 10 releases of eMatrix. This is the default setting. Databases using this setting are subject to the Oracle concurrency bug regarding non-indexed foreign keys.

- Index

The index setting results in an Oracle index being added to every column that is defined as a foreign key. The index setting results in an Oracle index being added to every column that is defined as a foreign key, and has no existing index. Overhead in both storage and performance is added since updates to foreign keys also require updates to their corresponding index. It is recommended that this option be used in development and test environments, where there is substantial benefit in the enforcement of foreign key constraints, and the negative storage/performance impact will not affect large numbers of users. When issuing the command to add indices to the system, you can specify the tablespace to use for the indexing operation.

- None

This option improves concurrency by removing non-indexed foreign key constraints, thus no additional Oracle index is required. This option eliminates the concurrency problem of foreign keys, and in fact, further improves system performance and scalability by eliminating the low-level integrity checks performed at the database level. It is recommended that this option be used once an application has been thoroughly tested and is rolled out to a large-scale production environment.

When the system is set with a constraint mode, not only are the changes made to the relevant columns, but also the setting is stored in the database so that all future operations including creation of new tables, running the index vault command, and upgrade, will use the selected mode.

Note that these options only affect the approximately dozen columns (among all matrix tables) that have a foreign constraint but not an index.

System Decimal Symbol

eMatrix relies on the MX_DECIMAL_SYMBOL setting to present real numbers in eMatrix desktop or Web Navigator, and other custom ADK programs including the Value Chain Portfolio applications. This setting must be synchronized with the Oracle database setting for NLS_LANG.

To ensure that these settings are synchronized, the following MQL command is available for use by System Administrators only:

```
set system decimal CHARACTER;
```

where CHARACTER can be . or , [period or comma].

This command sets the Oracle-expected decimal character in the database.

For example, to set the system decimal to a comma, use:

```
set system decimal ,;
```

When setting the system decimal character, be sure to use the setting that is implied by the database's NLS_LANG setting. The default setting is period (.). So, if the database setting for NLS_LANG is AMERICAN_AMERICA.WE8ISO8859P15, Oracle expects a period for the decimal symbol (as indicated by the territory setting of AMERICA) and eMatrix will convert as necessary, once the following command is run:

```
set system decimal .;
```

The system decimal symbol must be synchronized with the Oracle database setting for NLS_LANG. MX_DECIMAL_SYMBOL no longer needs to be synched with the other settings, but is used to indicate the user's preference for display.

When connecting to a database via any eMatrix product, users typically enter real numbers using the decimal character defined in their MX_DECIMAL_SYMBOL setting. Once the system decimal setting is in place, eMatrix will substitute the correct character as necessary when it displays the data or writes it to Oracle. For example, with the database configured with a period (.) decimal character and the user's MX_DECIMAL_SYMBOL set to a comma (,), eMatrix will always display numbers to that user with a comma and always write them to Oracle with a period, regardless of how the user enters the number. The same is true for the reverse; if the database is set to a comma and the user's preference is set to a period, numbers are displayed with a period but saved with a comma.

When exporting business objects, MX_DECIMAL_SYMBOL influences the format of real numbers written to the export file. Therefore, MX_DECIMAL_SYMBOL should be set in the same way by the user that imports the file, or errors will occur.

Refer to the *eMatrix Installation Guide* for more information on configuring Oracle for multiple language support.

Allowing Empty Strings for Object Names

The following system-wide setting is available so that System Administrators can enable/disable the creation of objects with empty name fields in MQL for all user sessions permanently:

```
set system emptyname [on|off]
```

By default, the setting is off, which means that empty names are not allowed. MQL and any other program code will issue an error if any attempt to make a business object have an empty name is made. It will also cause a warning to be issued if a query specifies an empty string (" ") for the name. (The query will not error out and will not abort any transaction going on.) If the setting is on, the system allows empty names for objects in MQL only.

Setting History Logging for the System

The following system-wide setting is available so that System Administrators can enable/disable history for all user sessions permanently:

```
set system history [on|off]
```

By default, history is on. When turned off, custom history records can be used on the operations where history is required, since it will be logged regardless of the global history setting.

Privileged Business Administrators

By default a business administrator can change context to any person without a password. This is to allow administrators and programs to perform operations on behalf of another user. A System Administrator can disable this system-wide “back door” security risk by issuing the following command:

```
set system privilegedbusinessadmin off;
```

After this command has been run, business administrators need a password when changing context to another person. Only system administrators can change context to any other person without a password.

A System Administrator can re-enable business administrators to change context without a password using:

```
set system privilegedbusinessadmin on;
```

System tidy

In replicated environments, when a user deletes a file checked into an object (via checkin overwrite or file delete), by default all other locations within that store maintain their copies of the now obsolete file. The file is deleted only when the administrator runs the `tidy store` command.

You can change the system behavior going forward such that all future file deletions occur at all locations by using:

```
set system tidy on;
```

Since this command changes future behavior and does not cleanup existing stores, you should then sync all stores, so all files are updated. Once done, the system will remain updated and tidy, until and unless system tidy is turned off.

Running with system tidy turned on may impact the performance of the delete file operation, depending on the number of locations and network performance at the time of the file deletion.

While there will be no obsolete files, ingested stores may need to be defragmented periodically even with system tidy on.

List Statement

The list system statement is used to display all system settings.

```
list system ;
```

Your output will be similar to::

```
History=On  
change vault update sets setting =Off  
DecimalSymbol=.  
TidyFiles=Off  
Foreign constraint setting = Normal  
privilegedBusinessAdmin=On
```

Print Statement

The Print statement prints the specified setting to the screen::

```
print system |changevault | ;  
|constraint | ;  
|decimal | ;  
|history | ;  
|privilegedbusinessadmin| ;  
|tidy | ;
```

For example::

```
print system changevault;
```

Your output will be similar to::

```
updateSets=Off
```

Validating the eMatrix Database

The MQL validate command enables you to check the correctness and integrity of the eMatrix database. Using an Oracle database, the validate command issues a series of SQL commands that perform variations of the Oracle “analyze table” construct. For DB2, the validate command issues the runstats or reorg command based on the level specified.

There are five levels (0-4 in ascending order) used to check the correctness and integrity of the eMatrix database. A validate schedule using a mix of levels can be established to ensure continuing validity of the database, while optimizing the length of time necessary for the verification.

- **Level 0**

Scans the database for flagrant discrepancies within the database bytes. It does not open or check objects.

- **Level 1**

Performs Level 0 and then scans the objects of the database with minimal verification. For example, if Level 1 recognizes a Person object, it ensures that it is named.

- **Level 2**

Deletes statistics on all tables.

- **Level 3**

Estimates statistics on all tables.

- **Level 4**

Computes statistics on all tables.

If validation errors are discovered, it does not mean the database is damaged, especially if they are detected in the higher level validations. These errors should be analyzed by an eMatrix Engineer to determine the severity of the inconsistencies and fix any issues that can be corrected.

The MQL syntax is:

```
validate [level NUMBER] [output FILENAME] [VALIDATE_ITEM { ,VALIDATE_ITEM}];
```

NUMBER is 0, 1, 2, 3, or 4

FILENAME is the file name to which the DB2 command is written (not used for Oracle).

VALIDATE_ITEM can be any of the following:

vault VAULT_NAME { ,VAULT_NAME }	
store STORE_NAME { ,STORE_NAME }	See Validate Store Statement in Chapter 5
store STORE_NAME file FILENAME	See Validate Store Statement in Chapter 5
index vault VAULT_NAME table TABLE_NAME	See Clearing and Maintaining Vaults in Chapter 4
process PROCESS_NAME { ,PROCESS_NAME }	See Validating a Process in Chapter 21
upgrade	See Installing a New Version of eMatrix in the <i>eMatrix Installation Guide</i> .
index NAME	See Validating an Index in Chapter 8

For vault and store, if no level number is specified, a Level 4 validation is performed. Ingested stores are the only type of store that can be validated. The same level validation can be performed on any combination and number of stores and vaults in one command. For example,

```
validate level 2 store 'Drawing Documentation', vault Engineering;
```

Using an Oracle Database

On Oracle, the MQL validate command results in various forms of the SQL ‘analyze table’ statement being executed against all tables for a given vault. The particular analyze option is determined by the level setting for validate:

- Level 0: analyze table xxx validate structure;
- Level 1: analyze table xxx validate structure cascade;
- Level 2: analyze table xxx delete statistics;
- Level 3: analyze table xxx estimate statistics;
- Level 4: analyze table xxx compute statistics;

For instance, the MQL command:

```
validate level 3 vault Actuators;
```

results in statistics being estimated on all tables in the Actuators vault.

Using a DB2 Database

For DB2, the validate command issues the runstats or reorg command based on the level specified:

- Level 0: runstats
- Level 1: runstats for indexes all
- Level 2: runstats and indexes all
- Level 3: reorgchk
- Level 4: reorg

It is important to note that validate level 4 uses the DB2 reorg command. Since reorg physically reorganizes tables and is resource intensive, only a Database Administrator should run level 4 validation.

A database backup is recommended before running level 4 validation.

Additionally for DB2, an “output” parameter is required for the validate command that specifies the file name to which the DB2 command is written, for example:

```
validate level 0 output actuators.bat vault Actuators;
```

The above validate command will create the actuators.bat file containing the runstats command for all tables in the Actuators vault.

You then run the batch file from the DB2 command line, which in turn will execute the appropriate DB2 commands.

See IBM DB2 documentation for more about the runstats, reorgchk, and reorg commands.

Maintaining and Monitoring Clients

Viewing User Session Information

Some of the tasks you must perform as the eMatrix System Administrator require that you shut down the eMatrix or system software. Before doing this (and perhaps at other times), you may need to know which users are currently using the eMatrix database. You can use the Sessions command to view a list of all current users.

```
sessions;
```

This command provides output of the form:

```
USERNAME MACHINENAME PROGRAMNAME  
USERNAME MACHINENAME PROGRAMNAME
```

PROGRAMNAME includes the path, executable name, and any command line options being used. For example:

```
peter PETESMACHINE c:\matrix\bin\winnt\mql.exe -k
```

If the executable was started with a shortcut on Windows NT, the PROGRAMNAME displayed is limited to the first 64 characters.

You must be logged in as the System Administrator to use the Sessions command. If you receive the following error message, access to the table that stores session information was not configured when the system was installed:

Table or View does not exist.

The Oracle Database Administrator must run the following SQL*Plus command:

```
GRANT SELECT ON "SYS"."V$_SESSION" TO "MATRIX";
```

The sessions command is not currently supported on DB2.

Error Log and Disk Space

eMatrix errors are automatically written to mxtrace.log, located by default in MATRIXHOME, or in the directory defined by MX_TRACE_FILE_PATH in the Matrix.ini file.

When a user that is not defined as a Business or System Administrator executes a program that issues a print program select code dump command (or executes the command in MQL), an error is posted in mxtrace.log. In RMI and EJB Collaboration Server environments this error does not occur. But since all PS Application Library programs and many custom programs use this command, the size of the mxtrace.log file may increase dramatically in a desktop environment that uses these programs regularly.

For that reason, periodic cleanup of the mxtrace.log file is recommended for both server and client machines.

Access Log

You can enable access logs that provide auditing of all access right grants and denials on a client's system. UNIX systems make use of the UNIX syslog(3) interface to log, write, or forward messages to designated files or users; on NT systems, the Application Event Log is used. Because these O/S logging facilities are utilized, existing auditing tools can be easily modified to include the eMatrix access log.

eMatrix checks user access definitions in the business object's governing Policy. Access can also be defined on Relationships (to/from/connect/disconnect, changetype, freeze, thaw, modifyattributes), Attributes (read, modify), Forms (viewform, modifyform) and Programs (execute) by assigning Rules. The access log includes information on where the access was granted—by virtue of the object being in which state in the Policy, or by the existence of what Rule, and the user being part of which Group or Role.

Note that an entry is added only when access is checked. This means that since access is never checked for a user defined as a System Administrator, no log output is generated for this type of user. Additionally, although access may be allowed, this does not infer that the action was successfully completed.

Enabling the Access Log

Access logging is enabled by adding the following line for Windows NT to ematrix.ini, bos.ini, or matrix.ini:

```
MX_ACCESS_LOG = true
```

For UNIX, set and export the setting as an environment variable in desktop client application startup script (matrix, MQL), startWebLogic.sh or rmireg.sh. For example:

```
MX_ACCESS_LOG = true
export MX_ACCESS_LOG
```

The default is false, meaning logging of access is off.

Log Output

Whether it's part of the UNIX syslog, or the NT Event Log, the information logged on each type of object varies as shown in the tables below. Capitalization in the formats indicate a substitution from eMatrix.

Access Log Data: Business Objects	
Access Description	Access on business objects is determined by policy and state.
Granted log format	POLICY::STATE::ACCESS allowed for USER[,AUTH] in GROUP/ROLE [as GRANTOR] on TYPE NAME REV in VAULT (based on policy)
Denied log format	POLICY::STATE:ACCESS denied for USER[,AUTH] on TYPE NAME REV in VAULT
Sample output	Production::Released::checkin allowed for Des in Designers on Assembly MTC1 A in Parts;

Access Log Data: Relationships	
Access Description	Access checks on relationships involve two steps. The first is to check if the requested access is allowed on the business objects to which the relationship is connected. These access checks, success or failure, will be logged as for business objects. If successful, the existence of an access rule assignment on the relationship type is checked. If the rule exists, success or failure will be logged as described below.
Granted log format	::RULE::ACCESS allowed by USER[,AUTH] in GROUP/ROLE [as GRANTOR] on RELTYPE (OID)
Denied log format	::RULE::ACCESS denied for USER[,AUTH] on RELTYPE (OID)
Sample output	Production::Released::todisconnect allowed for Des in Designers on Assembly MTC1 A in Parts; Production::Released::fromdisconnect allowed for Des in Designers on Assembly EZ45 A in Parts; ::DesignedRule::disconnect allowed for Des in Designers on AsDesigned

Access Log Data: Attributes	
Access Description	Similar to the behavior on relationships, attribute access is first checked on the business object or relationship to which the attribute is assigned. These access checks will be logged as described above. If successful, a second check is made against an optional access rule assigned to the attribute type. If the rule exists, success or failure will be logged as described below.
Granted log format	::RULE::ACCESS allowed for USER[,AUTH] in GROUP/ROLE [as GRANTOR] on ATTRIBUTE (OID)
Denied log format	::RULE::ACCESS denied for USER[,AUTH] on ATTRIBUTE (OID)
Sample output	Production::Released::modify allowed for Des in Designers on attribute Assembly MTC1 A in Parts; ::CostAttr::modify allowed by Des in Designers on TargetCost

Access Log Data: Programs and Forms	
Access Description	Programs and forms are nearly identical in how access checking is performed. An access rule can be assigned to a Program or Form object that is checked whenever a user attempts to execute the Program or open the Form. Such access checks will be logged as shown below.
Granted log format	::RULE::ACCESS allowed for USER[,AUTH] in GROUP/ROLE [as GRANTOR] on PROGRAM/FORM
Denied log format	::RULE::ACCESS denied for USER[,AUTH] on PROGRAM/FORM
Sample output	::MGRCount::execute allowed by Des in Designers on CountParts

Reading a UNIX Access Log

The Access Log output on UNIX goes to 'syslog.' The output destination for Syslog messages can be controlled by modifying the syslog.conf file which may be located in the /etc directory (it is not necessarily in a standard location on a server). Create a file to hold the messages and then insert a line into the file such as:

```
* .info /usr/USERNAME/AccessLogMessages
```

This results in successful access messages being written into a file named AccessLogMessages.

The file specified must exist before adding the entry in the syslog.conf file.

The UNIX system handles the messages, successful access and failed access, as *.info and *.notice respectively.

Successful access attempts will be recorded as priority LOG_INFO. Failed access attempts will be recorded as priority LOG_NOTICE.

In order get the new entry (in the syslog.conf file) recognized, the server must be restarted, or a kill command must be executed with the hangup option for the syslog process id. For example:

```
kill -HUP `cat /etc/syslog.pid`
```

NOTE: The syslog.pid file located in different areas on different machines.

Sample Output

```
Feb 21 21:58:42 gesun1 mxaccess[17387]: DOCUMENTS::WIP::read allowed for
Buju on Note BujuNote 0 in CM
Feb 21 21:58:45 gesun1 mxaccess[17387]: DOCUMENTS::WIP::demote allowed for Buju(owner)
on Note BujuNote 0 in CM
Feb 21 21:58:45 gesun1 mxaccess[17387]: DOCUMENTS::Planning::read allowed for Buju on
Note BujuNote 0 in CM
Feb 21 21:59:34 gesun1 last message repeated 1 time
```

A new entry is not added if the prior entry is the same. Instead, the number of times that the entry is repeated is entered, and then followed by a different entry.

Using the NT Event Viewer

The Event Viewer is used to view the access log on NT, as described in the procedure below.

To View the access log on NT

1. From the Start menu, choose Event Viewer from the **Programs>Administrative Tools** program.
2. Change from the System log to the Application log by choosing **Application** from the **Log** menu.
3. To view any of the access log entries, double click any of the entries whose **Source** is eMatrix.



4. You can scroll through the access log using the **Previous** and **Next** buttons.
5. Click **Close** when finished viewing the entries.

Monitoring Servers

Diagnostic tools have always been available for eMatrix servers using standard output. However, in the thin client environment, where the eMatrix Kernel runs inside a Web server, accessing trace information that is piped to stdout can be awkward. Enhanced server diagnostic tools allow the following trace information to be sent either to a file or to standard output (the “destination”):

- SQL output
- MQL trace: program execution, including both Tcl and Java program objects
- VERBOSE trace of client/server dispatches
- FTP and FTPS communication
- SMTP communication
- LDAP communication
- Trigger execution
- Workflow debugging
- User-defined trace types

All tracing mechanisms for the various eMatrix Kernel operations can be turned on and off for all sessions via environment variables in a UNIX startup script or .ini file settings, or for either all sessions or single sessions via MQL or the ADK.

Enabling Tracing

Traditionally eMatrix tracing has been enabled via environment variables in a startup script (such as startWeblogic, rmireg or Matrix) or Windows .ini file, and therefore affected all sessions. Refer to the eMatrix Installation Guide for more information.

However, since these settings enable tracing of activities for all user sessions, using .ini settings for starting/stopping tracing should be restricted to those cases where it is necessary to trace startup behavior. It has two distinct disadvantages that suggest it should be avoided unless absolutely necessary:

- It requires you to stop the server to reset .ini settings.
- It affects all users. Tracing does slow the system down, especially tracing low-level operations (verbose, sql) across all sessions.

You can enable both all-session and single-session tracing via MQL or the ADK, which allows more flexibility.

All-session vs. single-session tracing

The directory that will hold all tracing files is defined via the MX_TRACE_FILE_PATH setting in the .ini file or startup script. At any time there is only one file (or output stream) used to receive the tracing output for all types of all-session tracing. The filename used is specified in the .ini or startup file as the value of an MX_*_TRACE setting, or via the MQL/ADK interfaces using the `filename` modifier (as described in [Enabling trace tools via MQL](#) and [ADK interfaces](#)). The trace settings are read in the following order, and the first one found with a filename specified establishes the output filename for all of them:

`MX_TRIGGER_TRACE`

```

MX_VERBOSE_TRACE
MX_SQL_TRACE
MX_MQL_TRACE
MX_FTP_TRACE
MX_WORKFLOW_TRACE
MX_SMTP_TRACE
MX_LDAP_TRACE
MX_TIMER_TRACE

```

For example, if a filename is specified for MX_VERBOSE_TRACE and MX_SQL_TRACE, and if MX_TRIGGER_TRACE is set to true, the output for all three kinds of tracing will go to the stdout. If MX_SQL_TRACE and MX_MQL_TRACE are both set to a filename, the output for both will go to the file specified for MX_SQL_TRACE.

A list of single-session trace filenames, one for each session that requests some kind of tracing, is also maintained by the kernel. Single-session trace files are specified in the MQL/ADK commands that enable the tracing (as described in [Enabling trace tools via MQL](#), and [ADK interfaces](#)). Once a filename has been specified for a specific single-session trace type, it will be used for all single-session tracing that is requested by the same session. Similarly, tracing requests from other distinct sessions will have their own filename on the list so that distinct single-session traces will not interfere with each other.

One exception to the uniformity is that VERBOSE tracing, which traces client-server calls, is available for all sessions only. It can, however, be started and stopped via MQL/ADK. So, if verbose tracing and some sort of single-session tracing are both enabled with a filename, two separate files will be created.

All-session traces are maintained in a single file, established by the setting of the filename of the most predominant type of tracing (as listed above). Single-session traces are also maintained in a single file, though separate from all-session traces. The single-session filename is determined by the first filename specified, no matter what the trace type. These files are located in MX_TRACE_FILE_PATH.

The following table shows the effect of having tracing turned on for all sessions, single sessions, or both. Note that with the exception of verbose tracing, the single-session trace filename wins.

Tracing type	Turned on for all sessions	Turned on for this session	File used
Verbose	Yes	n/a	All-session file
	No	n/a	none
All Others	Yes	Yes	Single-session file
	Yes	No	All-session file
	No	Yes	Single-session file
	No	No	none

Enabling trace tools via MQL

All types of tracing can also be turned on (and off) from a running session of the Kernel using the following MQL commands:

```
trace type TYPE |filename FILENAME [all]|
|on [all]|
|off [all]|
|[not] full|
|text STRING|
```

Each clause is described in the sections that follow.

type TYPE clause

TYPE is the type of tracing to enable, and is one of the following

sql
mql
verbose
ftp
smtp
ldap
trigger
workflow
store
OTHER_TYPE

Particular care should be taken in the use of SQL and VERBOSE tracing. Both are low level, and liable to generate a large amount of output data, which will affect performance.

The keyword `store` can be used with the `trace type` command, so that when synchronizing or purging stores a log file can be created that indicates the success or failure of every file and business object.

`OTHER_TYPE` allows the definition of a user-defined tracing type, for example, “MYTRACE”. Refer to [text STRING clause for user-defined trace types](#) for more information.

file FILENAME clause

Specifying a file with the `file FILENAME` clause turns the type of tracing specified on, and redirects the output to the specified file. There is no need to use the keyword “on”. If a filename has already been specified for another type of tracing within the current session, that filename will be used, and the one specified here will be ignored, but the tracing will be enabled.

If the filename specified for any tracing already exists in the directory `MX_TRACE_FILE_PATH`, that existing file will be copied to a backup whose name is

constructed by prepending a time-specific prefix to the filename, in the form “yyymmddhhmmss__FILENAME.”

on clause

The `on` modifier will turn the specified tracing on and send it to a previously specified trace file, or if there is no previously specified trace file, it will send the trace information to `stdout`. By default, tracing is started for the current session only, and will NOT record activities invoked by other sessions.

all clause

You can include the `all` clause when you want to record trace information of the specified type for all sessions. By default, you only enable it for the current session.

One exception is that `VERBOSE` tracing, which traces client-server calls, is available for all sessions only.

Refer to [All-session vs. single-session tracing](#) for more information.

off clause

The `off` clause turns the specified type of tracing off. If other tracing types had been turned on, they will stay on. Tracing that is turned on via .ini file settings or using the `all` keyword can only be turned off using `all`. For example:

```
trace type SQL off all;
```

When all types of tracing directed at the same file are turned off, the file is closed. In order to resume tracing to a file, any subsequent command must specify a filename. If it doesn’t, tracing will be resumed with `stdout` as the destination.

[not]full clause

Trace output includes timestamp information, which you can turn off with the `notfull` clause. You can also use `!full`. To re-enable, use the `full` clause.

text STRING clause for user-defined trace types

Programmers can embed tracing messages in their implementation code to provide strings to be output to the trace file of type `OTHER_TYPE` using the following:

```
trace type OTHER_TYPE text STRING;
```

With these types of messages within programs, you would then enable the tracing with one of the following:

```
trace type OTHER_TYPE on;  
trace type OTHER_TYPE filename FILENAME;
```

Refer to [Examples](#) for sample output.

ADK interfaces

Two sets of interfaces are provided that can be used in your Java application to output tracing messages of any sort to the eMatrix trace files. These messages will be output only

at runtime if the corresponding trace TYPE has been turned on. The tracing interfaces available to Java programmers are:

- as enhancements that have been made to the object-oriented MatrixLogWriter, which may be most appropriate in complex Java applications where the MatrixLogWriter object can be instantiated and maintained to control tracing across a set of interacting contexts/sessions.
- through the current Context object. These procedural methods may be most appropriate for use in implementations which do not have a central controlling application, such as JSPs and eMatrix Java program objects.

MatrixLogWriter methods

With the object-oriented MatrixLogWriter interface, you turn tracing on by instantiating a MatrixLogWriter object with one of two constructors:

```
public MatrixLogWriter(Context context)
```

Turn tracing on for all sessions with type=LOGWRITER, filename=eMatrix.log

```
@param Context context: the current active context  
@return none
```

```
public MatrixLogWriter(Context context, String filename,  
String type, boolean allFlag)
```

Turn tracing of specified type on/off for this/all sessions

```
@param Context context: the current active context  
@param String filename: name of file for tracing output  
@param String type: type of tracing desired (all caps: MQL, SQL, MINE)  
@param boolean onFlag: true to turn trace on, false to turn it off  
@param boolean allFlag: true for all sessions; false for this session only  
@exception MatrixException when context cannot be established
```

Once it is instantiated, the trace file is open, and writing to it consists of simply calling the write method. The timestamp and label (for the type) will be formatted and prepended to the message.

```
public void write(String message) throws java.io.IOException  
public void close() throws java.io.IOException
```

It is important to remember to call the close function before destroying the MatrixLogWriter object or allowing it to be destroyed by going out of scope. Failure to do so will result in the process holding onto the open file descriptor, so the file cannot be deleted or renamed until the process (your server) terminates.

Context methods

ADK interfaces are also provided that correspond to the MQL commands in the preceding section. These interfaces are provided as methods on the current Context object:

```
public void setTrace(String filename,  
String type,
```

```

        boolean onFlag,
        boolean allFlag) throws MatrixException

@param String filename: name of file for tracing output
@param String type: type of tracing desired (MQL, SQL..., MYTRACE)
@param boolean onFlag: true turns trace on, false turns it off
@param boolean allFlag: true traces all sessions;
                           false traces this session only

@exception MatrixException when context cannot be established
@return none

```

As with the MatrixLogWriter interface, it is important to remember to turn all traces off when you have collected the information you require, especially when you are running your traces against a production server. Tracing consumes resources, and unclosed file descriptors are among them.

```

public void printTrace(String type,
                      String message) throws MatrixException

@param String type: type of tracing desired (MQL, SQL..., MYTRACE)
@param String message: message to be printed to trace file

@exception MatrixException when context cannot be established
@return none

```

Output

All trace entries have a standard prefix consisting of a fixed-format timestamp (without the date), the tracing TYPE, the thread ID and a trace message for each line of output. The prefix will always be 23 characters long, followed by a space, as follows:

21:51:39.795 TYPE t@284 <tracing output>

So, the trace message always begins in column 25. For example:

02:06:59.339 MQL t@354 Start MQLCommand: exec prog printContext

Output columns are as follows:

- Column 1 through 12 contains the timestamp: hour (0-24), minute, second, millisecond
- Column 14 through 17 contains the tracing type: always 4 characters (padded or truncated, such as: “SQL”, “MQL” “MYTR”)
- Column 19 through 23 contains the thread id: t@ followed by 3 digits
- Column 25 - tracing message generated by eMatrix Kernel, by MQL ‘trace type text’, or by ADK calls to `printTrace()`.

This table summarizes the output provided for the various traces. Detailed examples are in the following sections.

Trace Type	Output
SQL	Select, update, insert, delete, rollback and commit commands issued to Oracle. All output is preceded by thread ID.

Trace Type	Output
MQL	Program name and arguments (Tcl or JPO), MQL commands issued by those programs (Tcl MQL command or JPO mqlCommand interface), total time for program execution, and execution time for each MQL command. All output is preceded by thread ID. Session ID will also be printed for Program names and commands executed via the mqlCommand interface*.
*	The session ID is a long string, which would make for hard-to-read output if we were to prepend every command with it. The thread ID is usually sufficient. Therefore, we output session id only with certain server calls, and provide the corresponding thread id, which will identify subsequent trace messages associated to that thread, hence that session.
FTP	Open, read, write, close requests/responses to/from FTP server
SMTP	Open, send requests/responses to/from SMTP server
LDAP	Connect and informational requests/responses to/from LDAP server. All output is preceded by thread ID
Trigger	Event and type (check, override, action) for each trigger fired, and the name of the program that is run.
Workflow	Processing data for workflow events.
<other>	Whatever the implementor has specified via 'trace type TYPE text STRING' commands or corresponding ADK calls.

Tracing and Debugging

User-defined tracing should be considered at the time that you are developing your implementation. Keep in mind the following:

- Because tracing can be turned off, you should consider whether you should leave a few judiciously-placed calls to printTrace for production level diagnosis and performance checks. They will be inactive until or unless you explicitly turn on tracing of the corresponding type. The emphasis here is on “few” and “judiciously-placed,” since even inactive calls have to do some checking to discover they are inactive. For example, putting calls to MatrixLogWriter to write “entering program” and “exiting program” for every method in a complex client application, would be frivolous, since each represents an additional call to the server where the tracing flags are maintained. However, the inclusion of a few tracing calls before and/or after some complex operations can streamline downstream troubleshooting considerably.
- Matrix merges multiple trace types into a single file, and may do so for multiple (all) sessions or threads. The inclusion of explicit TYPE tags in each label allows you to use OS tools to extract records for a single trace type into its own file. Similarly, the inclusion of the thread identifier in each label allows you to extract/segregate the activities of all types for a single session or thread.
- You should also carefully consider the placement of performance traces, since the mere fact of turning them on affects the very thing they are intended to measure. For performance analysis, the fixed format lends itself well to parsing the log files and measuring time spent in certain programs, etc.

The following example shows how to turn single-session tracing on for SQL, MQL and a user-defined tracing type ‘MYTRACE’, followed by the resulting text file myExample.log:

```
MQL<5>trace type SQL filename myExample.log;
MQL<6>trace type MQL on;
MQL<7>trace type MYTRACE on;
MQL<8>execute program testTrace;
>>>>>>>> myExample.log <<<<<<<<<<<<
```



```
2001349-22:12:27.740 SQL t@284 select * from mxVer6 where mxOid=:va
2001349-22:12:27.740 SQL t@284      :va=1
2001349-22:12:27.740 SQL t@284 -->select
2001349-22:12:27.740 MQL t@284 Session: mx3c1d5d9a16234334
2001349-22:12:27.740 SQL t@284 select * from mxVer6 where mxOid=:va
2001349-22:12:27.740 SQL t@284      :va=1
2001349-22:12:27.740 SQL t@284 -->select
2001349-22:12:27.740 MQL t@284 Program: testTrace
2001349-22:12:27.740 MQL t@284      args:
2001349-22:12:27.740 MQL t@284 allocating non-pool TCL interpreter
2001349-22:12:27.940 MQL t@284 Begin TCL Session: mx3c1d5d9a16234334

2001349-22:12:27.970 MQL t@284 mql: trace type MYTRACE text Start checkpoint: Temp
query rev=last
2001349-22:12:27.970 MYTR t@284 Start checkpoint: Temp query rev=last
2001349-22:12:27.970 MQL t@284 mql time 0.000
2001349-22:12:27.970 MQL t@284 mql: temp query bus * * * where 'revision == last'
2001349-22:12:27.970 SQL t@284 select * from mxVer6 where mxOid=:va
2001349-22:12:27.970 SQL t@284      :va=1
2001349-22:12:27.970 SQL t@284 -->select
2001349-22:12:27.970 SQL t@284 select * from lxBO_4a7a4a03
2001349-22:12:27.970 SQL t@284 -->select
...
2001349-22:12:27.980 SQL t@284 -->select
2001349-22:12:28.060 SQL t@284 select * from lxBO_9faf941c
2001349-22:12:28.080 SQL t@284 -->select
...
2001349-22:12:28.160 SQL t@284 -->select
2001349-22:12:28.521 SQL t@284 select * from lxBO_77298049
2001349-22:12:28.521 SQL t@284 -->select
2001349-22:12:28.561 MQL t@284 mql time 0.591
2001349-22:12:28.571 MQL t@284 mql: trace type MYTRACE text End checkpoint: After temp
query
2001349-22:12:28.571 MYTR t@284 End checkpoint: After temp query
2001349-22:12:28.571 MQL t@284 mql time 0.000
2001349-22:12:28.571 MQL t@284 TCL Results:
2001349-22:12:28.571 MQL t@284 END TCL time: 0.631 (0.591 in MQL)
```

```
2001349-22:12:28.571 MQL t@284 End Program:
```

Tracing with Tcl

The following is the definition of a Tcl program that might have been written to time the performance of a particular query.

```
tcl;
eval {
# This program
    set sStart [clock clicks]
    mql trace type MYTRACE text "Start checkpoint: Temp query rev=last"
    mql temp query bus * * * where 'revision == last'
    set sEnd [clock clicks]
    set sDiff [expr [expr 1.0 *[expr $sEnd -$sStart] / 1000]]
    mql trace type MYTRACE text "End checkpoint: After temp query $sDiff"
}
```

The time calculations are rendered somewhat redundant now since the new tracing output provides the timestamps:

```
2001354-22:34:16.327 MYTR t@360 Start checkpoint: Temp query rev=last
2001354-22:34:16.978 MYTR t@360 End checkpoint: After temp query 0.651
```

Even more so if you turn on MQL tracing, which prints out the time for the program execution:

```
2001354-22:37:10.177 MQL t@360 Session: mx3c23f9c4b94ee473
2001354-22:37:10.177 MQL t@360 Program: testTrace
2001354-22:37:10.177 MQL t@360 args:
2001354-22:37:10.187 MQL t@360 allocating non-pool TCL interpreter
2001354-22:37:10.387 MQL t@360 Begin TCL Session: mx3c23f9c4b94ee473

2001354-22:37:10.417 MQL t@360 mql: trace type MYTRACE text Start checkpoint: Temp
query rev=last
2001354-22:37:10.417 MYTR t@360 Start checkpoint: Temp query rev=last
2001354-22:37:10.417 MQL t@360 mql time 0.000
2001354-22:37:10.417 MQL t@360 mql: temp query bus * * * where 'revision == last'
2001354-22:37:11.259 MQL t@360 mql time 0.842
2001354-22:37:11.259 MQL t@360 mql: trace type MYTRACE text End checkpoint: After temp
query 0.842
2001354-22:37:11.259 MYTR t@360 End checkpoint: After temp query 0.842
2001354-22:37:11.259 MQL t@360 mql time 0.000
2001354-22:37:11.259 MQL t@360 TCL Results:
2001354-22:37:11.259 MQL t@360 END TCL time: 0.872 (0.842 in MQL)
2001354-22:37:11.259 MQL t@360 End Program:
```

Tracing with MatrixLogWriter

This example shows how MatrixLogWriter can be constructed with no arguments to write to eMatrix.log with type=LOGWRITER, or can be customized to write to a file of your choice with a tracing type of your choice:

```
{
    // this will write to default file (eMatrix.log) as LOGW
    MatrixLogWriter writer1 = new MatrixLogWriter(getContext());
    String message = "Message #1";
    writer1.write(message, 0, message.length());

    // this will write to myLOG1.log
```

```

MatrixLogWriter writer2 = new MatrixLogWriter(getContext(),
                                              "myLOG1.log",
                                              "MYTRACE", false);
message = "Message #2";
writer2.write(message, 0, message.length());
writer2.close();
// this will write to myLOG2.log
writer2 = new MatrixLogWriter(getContext(), "myLOG2.log",
                               "myother", false);
message = "Message #3";
writer2.write(message, 0, message.length());
writer2.close();
// append message to default file
writer1.write("Message #4");
writer1.close();
}
catch ( java.io.IOException e )
{
    // nothing
}

```

Tracing with Context.setTrace(), Context.printTrace()

Similarly, this example shows how to use Context::setTrace and Context::printTrace. It is rather contrived, but it shows the relationship of the single-session and all-session targets:

```

// In the following, three traces are turned on for all sessions,
// Then the same traces are turned on for 'this' session.
// The VERBOSE trace will stay stuck to the all sessions file,
// but the SQL and MQL traces for this session will be diverted
// to the local file for this session

// Then the local traces are turned off again allowing all the
// trace to go to the all sessions file again.

// Finally, the local (SQL,MQL) traces are turned back on, but
// directed to a different local file.

// A search for "TRACE" in the output files will show where
// the 'trace type xxx text ...' messages end up. Each one notes
// in square brackets where it SHOULD end up.

// Note that:
//     - VERBOSE always goes to ALL
//     - Single session tracing wins over all session tracing
//     - some of the traces (marked [NONE]) don't go anywhere,
//           because there's no trace file of the appropriate type open

// send traces for all sessions to all1.log
execCommand ("trace type VERBOSE filename all1.log all", resultList);
execCommand("exec prog printContext", resultList);
execCommand("trace type mql on all", resultList);
execCommand("trace type sql on all", resultList);
execCommand("trace type verbose text '[ALL1] TRACES ON FOR ALL1'", resultList);
execCommand("trace type sql text '[ALL1] TRACES ON FOR ALL1'", resultList);
execCommand("exec prog printContext", resultList);

// send traces for this session to local1

```

```

execCommand("trace type sql filename local1.log", resultList);
execCommand("trace type mql on", resultList);
execCommand("trace type verbose on", resultList);
execCommand("trace type verbose text '[ALL1] TRACES ON FOR LOCAL1'", 
resultList);
execCommand("trace type sql text '[LOCAL1] TRACES ON FOR LOCAL1'", resultList);
execCommand("exec prog printContext", resultList);

// close traces to all1
execCommand("trace type mql off all", resultList);
execCommand("trace type sql off all", resultList);
execCommand("trace type verbose off all", resultList);
execCommand("trace type verbose text '[NONE] TRACES OFF FOR ALL1'", resultList);
execCommand("trace type sql text '[LOCAL1] TRACES OFF FOR ALL1'", resultList);
execCommand("exec prog printContext", resultList);

// close traces to local1
execCommand("trace type mql off", resultList);
execCommand("trace type sql off", resultList);
execCommand("trace type verbose text '[NONE] ALL BUT VERBOSE OFF FOR LOCAL1'", 
resultList);
execCommand("trace type sql text '[NONE] ALL BUT VERBOSE OFF FOR LOCAL1'", 
resultList);
execCommand("trace type verbose off", resultList);
execCommand("trace type verbose text '[NONE] TRACES OFF FOR LOCAL1'", 
resultList);
execCommand("trace type sql text '[NONE] TRACES OFF FOR LOCAL1'", resultList);
execCommand("exec prog printContext", resultList);

// send traces for this sessions to local2.log
execCommand("trace type mql filename local2.log", resultList);
execCommand("trace type sql on", resultList);
execCommand("trace type verbose text '[NONE] TRACES ON FOR LOCAL2'", 
resultList);
execCommand("trace type sql text '[LOCAL2] TRACES ON FOR LOCAL2'", resultList);
// close traces for local2.log
execCommand("trace type sql off", resultList);
execCommand("trace type mql off", resultList);
execCommand("trace type verbose text '[NONE] TRACES OFF FOR LOCAL2'", 
resultList);
execCommand("trace type sql text '[NONE] TRACES OFF FOR LOCAL2'", resultList);
execCommand("exec prog printContext", resultList);
}

```

ADK Session Monitoring

Sessions that are connected to eMatrix via the ADK can be monitored to report information such as user names, cache sizes, object IDs, and the like. These statistics are useful for debugging ADK processes and monitoring memory usage.

The following features are available for monitoring ADK sessions:

- `MX_VERBOSE_PARAM_TRACE` variable
- `monitor context` MQL command (RMI and EJB only)
- `monitor memory` MQL command

Verbose Tracing

MX_VERBOSE_PARAM_TRACE when set to true can show key ADK call parameters in verbose tracing. On Windows, you add this variable to the `ematrix.ini` file; on UNIX, add and export it as an environment variable to `rmireg.sh`.

When you set MX_VERBOSE_PARAM_TRACE to TRUE or `ematrix.log` (your log file name), tracing output includes additional ADK call parameters, such as:

- The user logging in, which is identified in entries including `allocExternalContext.bosInterface` and `reset.bosContent`.
- Entries listing “input params” and “output params”, which contain parameter information such as user, vault, object name, language, command, etc.
- “stateful dispatch” messages, which are followed by a report of the session name.
- The `open.bosBusinessObject` call, which also reports the input object-id and output TNR.
- The `open.bosQuery` call “input params” message, which also reports query details.
- The `evaluate.bosQuery` call “output params” message, which reports the number of objects returned.

The above information can be used to identify specific ADK calls that cause problems within the core.

Note that when MX_VERBOSE_PARAM_TRACE is set to TRUE or `ematrix.log`, verbose logging is created even if MX_VERBOSE_TRACE is not set.

The example below reports information about input params and the session ID:

```
[15:23:34.539 --- eMatrix@HEWEY 0 : t@2224: stateless dispatch for
allocExternalContext.bosInterface]
[15:23:34.539 --- eMatrix@HEWEY 0 : t@2224:    input params:
sessionId=PUFpXp1kjpqvcIIxhXAfhQqZdNlR89L2BVFBzjNn2l3qufgw28kt|-2636696131106167120/
167839130/6/7001/7001/7002/7002/7001/-1, user=Test Everything, vault=, lang=en-us,
tz=America/Los_Angeles]
[15:23:34.539 --- eMatrix@HEWEY 0 : t@2224: dispatch complete]
```

The example below reports a business object being opened, including input and output params along with the object ID of the business object being opened:

```
[15:23:51.143 --- eMatrix@HEWEY 0 : t@2224: stateless dispatch for
open.bosBusinessObject]
[15:23:51.143 --- eMatrix@HEWEY 0 : t@2224: allocate context for session
PUFpXp1kjpqvcIIxhXAfhQqZdNlR89L2BVFBzjNn2l3qufgw28kt|-2636696131106167120/167839130/6/
7001/7001/7002/7002/7001/-1]
[15:23:51.163 --- eMatrix@HEWEY 0 : t@2224:    input params:
id=57622.20620.20359.61650]
[15:23:51.173 --- eMatrix@HEWEY 0 : t@2224:    output params: returnVal
objectid=57622.20620.20359.61650, type=Person, name=Test Everything, revision=-,
vault=eService Administration]
[15:23:51.173 --- eMatrix@HEWEY 0 : t@2224: dispatch complete]
```

The following example of a query evaluation reports input parameters that describe the scope of the query, and an output parameter that indicates the number of objects returned for the query. (If the timestamps indicate the process took a long time, a large number of objects returned could explain the long timespan.)

```
[15:25:28.623 --- eMatrix@HEWEY 0 : t@2224: stateless dispatch for evaluate.bosQuery]
[15:25:28.683 --- eMatrix@HEWEY 0 : t@2224: allocate context for session
PUFpXp1kjpqvcIIxhXAfhQqZdNlR89L2BVFBzjNn2l3qufgw28kt|-2636696131106167120/167839130/6/
```

```
7001/7001/7002/7001/-1]
[15:25:28.733 --- eMatrix@HEWEY 0 : t@2224:    input params: name=, query type=Part,
name=*, revision=*, lattice=*, owner=*, where=, limit=1001, expandTypes=true]
[15:25:28.853 --- eMatrix@HEWEY 0 : t@2224:    output params: returnVal objects=176]
[15:25:28.853 --- eMatrix@HEWEY 0 : t@2224: dispatch complete]
```

Monitoring Context Objects

The `monitor context` MQL command is used to count and list currently registered `bosContext` objects, which map to ADK Context objects and give eMatrix core statistics:

```
monitor context [SESSION-ID] [set|!set] [terse];
```

`SESSION-ID` is used to limit the display of session information to the context object for the specified session ID. If not used, all sessions are reported.

The `[set | !set]` option limits the display of session information to contexts that are marked as “set” or “not set” respectively. (The `ADK context.shutdown()` method will mark a context “not set.”) Use this option only when the session ID is not specified.

`terse` displays only cumulative statistics and eMatrix statistics, not individual sessions.

The `monitor context` command can display the following information per session:

- number of contexts
- idle vs. active session
- total cached bytes
- session ID
- context set/logged in status
- username
- timestamp for and name of last ADK call and thread on which it executed
- transaction status (for example: active, mode, savepoints, etc.)
- estimate of context-specific and admin cache size in use, if possible, for each active session

The command displays the following environment information for all sessions:

- Total number of sessions
- Administration cache size
- Total session cache size
- Pooled session cache size

Following is sample output for the `monitor context` command, taken using an ADK program that mimics eMatrix desktop MQL functionality. Note that the string named after the session name (`current`) indicates the context corresponding to the current user session.

```
mql>monitor context
Admin cache: 1260859 bytes
Pooled session cache: 0 bytes

4 context objects
```

```

Session PUF93121k9lAjAH0hy0O2WOZCYOhggu2dvWd0owsfT9DDHzH1I5P |
-263669613110616712
0/167839130/6/7001/7001/7002/7002/7001/-1
    User: 'Test Everything' logged in
    Vault: 'eService Sample'
    Last: t@2208, select.bosBusinessObject
    Last recorded cache size: 0
    idle: 14 minutes 52 seconds
    0 active sessions

Session mx1027692483622676970837 (current)
    User: 'creator' logged in
    Vault: 'ADMINISTRATION'
    Last: t@2104, executeCmd.bosMQLCommand
    Last recorded cache size: 0
    Idle: 0 seconds
    1 active session
        session 0
            transaction active,readonly,wait
            0 cached entries, 0 bytes

Session mx10277030735012117155733
    User: 'creator' logged in
    Vault: 'ADMINISTRATION'
    Last: t@1940, executeCmd.bosMQLCommand
    Last recorded cache size: 505579
    idle: 5 seconds
    2 active sessions
        session 0
            transaction active,update,wait
            savepoint savel
            3 cached entries, 3192 bytes
        session 1
            transaction active,update,wait
            3 cached entries, 502387 bytes

Session mx1027703338082-1990050644
    User: 'creator' logged out
    Vault: 'ADMINISTRATION'
    Last: t@2284, executeCmd.bosMQLCommand
    Last recorded cache size: 0
    idle: 11 minutes 39 seconds
    0 active sessions

Total cache size: 505579 bytes

```

Only users with System Administrator privileges can execute the monitor context command.

Notes

- If a context is currently executing at the time another context issues the monitor context command, output for the active session will resemble the following:

```
Session PUGpAOiBm30cnMK5Bk27QcQYjcm2iq2UwMQzxh9KGjTTddp52xkr | -263669613110616712
```

```
0/167839130/6/7001/7001/7002/7002/7001/-1
```

```
User: 'Test Everything' logged in
Vault: 'eService Sample'
Last: t@1956, executeCmd.bosMQLCommand
Active: 2 seconds
Last recorded cache size: 0 (update requested; reissue monitor context command)
*** Cannot report session stats - session is active ***
```

The above sample shows the “Active” time for the session, and a warning message states “...session is active.”

- In the interest of thread safety, monitor context processing reports only what is safe to report. Access to cache and transaction information during monitor context processing is done in a thread-safe fashion so it does not impact system performance and stability of other sessions and the system as a whole.

Monitoring Memory

The `monitor memory` command issues memory statistics for the RMI server:

```
monitor memory;
```

The monitor memory command is available only on NT and HP platforms.

Example output for an RMI server on NT:

```
Used heap 6667360 bytes, free heap 1182488 bytes.
```

Example output for an RMI server on HP:

```
free memory is 20064 pages, 82182144 bytes
This process is using 56705024 bytes of RAM
This process is using 477429760 bytes of VM
This process is using 12554240 bytes of data
```

Only users with System Administrators privileges can execute the monitor memory command.

Developing a Backup Strategy

Because there are a number of factors that can cause a database failure, including power loss, hardware failure, natural disasters, and human error, it is important that you develop both a backup and a recovery plan to protect your eMatrix database. It is not enough that you *develop* a recovery plan, however. You must also test that recovery plan to ensure that it is adequate before your data is compromised. Finding out that your recovery plan is inadequate after you have already lost your data will not do you much good. Also, testing of the recovery plan may indicate changes you need to make to your backup strategy.

It is highly recommended that inventories of all stores are performed nightly as part of the backup procedure. Refer to [Inventory Store Statement](#) for details.

Refer to the *eMatrix System Manager Guide* for more information on developing a backup strategy.

4

Working With Vaults

Vault Defined	112
Kinds of Vaults.....	113
Business Object Vaults.....	113
Administration Vault.....	113
Defining a Vault.....	114
Description Clause.....	114
Icon Clause.....	115
Vault Types.....	115
Tablespace Clause	116
Indexspace Clause	116
Server Clause	116
Interface Clause.....	116
File Clause.....	116
Map Clause.....	117
Hidden Clause	117
Property Clause	117
Modifying a Vault Definition.....	118
Modifying a Vault Definition	118
Clearing and Maintaining Vaults	119
Clearing Vaults	119
Indexing Vaults	119
Fixing Fragmented Vaults	120
Updating Sets With Change Vault.....	120
Printing a Vault Definition	121
Deleting a Vault.....	122

Vault Defined

A *vault* is a grouping of similar objects within the eMatrix database, as well as a storage location for metadata which identifies those objects. For example, in an insurance company, a vault might contain insurance forms of a similar type or from a single geographical area. In an engineering environment, a vault might contain all objects related to a particular project or family of products. In a bank, a vault might include all accounts of a particular type or loans made during a particular period.

The Business Administrator determines what the vault is for, while the System Administrator defines where the vault is located on the network. Vaults should use actual host and path names, not mounted directories. Paths must be exported on the host to all users who require access to the vaults.

Information on how vaults are defined, modified, viewed, maintained, and deleted is presented in the sections that follow.

You must be a System Administrator to access vaults. (Refer also to your System Manager Guide.)

Kinds of Vaults

There are two kinds of vaults:

- Business object vaults (you can have many)
- An Administration vault (only one exists)

Business Object Vaults

Business object vaults are used to organize business objects within your database. How you do so will depend on the types of objects you use and the relationships they have to one another.

Vaults contain metadata (information about objects), while stores contain the application files associated with business objects.

All vaults contain a complete set of eMatrix definitions. These definitions identify the characteristics of items such as persons, roles, types, formats, etc. When you make changes to a definition (such as add, modify, or delete), all definition copies must be updated to reflect the change. This update of the vaults occurs simultaneously if all the copies are available. If any of the copies are not available (a vault is not available), you cannot alter the definitions. This prevents partial alteration of the eMatrix definitions.

For example, assume you want to add a new format definition. After you enter the Add Format statement, eMatrix will attempt to add the definition. If the definition is valid (no errors), all copies of the eMatrix definitions are changed to include this new format. But assume that a vault resides on a host that is currently offline. In this case, no changes to the definitions are made. If changes were allowed, the one vault would not be updated to contain the change. Therefore, you should ensure that all defined vaults are available before modifying the eMatrix definitions.

Administration Vault

The Administration vault is used for administrative purposes only and serves as the master definition vault. It is created automatically when eMatrix is installed on your system. Unlike other vaults, the Administration vault is not listed among the available vaults when a person uses the Vault Chooser to specify a vault when setting context, performing queries, or creating new objects in eMatrix Navigator or Web Navigator. The Administration vault is used for definitions only. You cannot use it for storing business objects.

Defining a Vault

Use the Add Vault statement to define a vault:

```
add vault NAME [ADD_ITEM {ADD_ITEM}] ;
```

NAME is the name of the vault you are creating. All vaults must have a unique name in the database. The name can be up to 127 characters long and can contain spaces. Since you have this flexibility in naming the vault, you should assign a name that has meaning to both you and the users.

For example, each of the following is a valid vault name:

Northeast Regional Area
Health Insurance Records
Housing Projects
Customer Service Department

ADD_ITEM is an Add Vault clause that provides more information about the vault you are creating. Although none of the clauses is required to make the vault usable, they are used to define a vault location other than the current default host and path. In addition, the clauses can help users understand the purpose of the vault.

The Add Vault clauses are:

description VALUE
icon FILENAME
indexspace SPACE
tablespace SPACE
server SERVERNAME
interface LIBRARYFULLPATH
file MAPFILENAME
map STRING
[! not]hidden
property NAME [to ADMINTYPE NAME] [value STRING]

Each clause and the arguments they use are discussed in the sections that follow.

Description Clause

The Description clause of the Add Vault statement can provide general information about the functions covered by the vault. It can also help point out subtle differences between vaults to the user. If a user is assigned the wrong vault, s/he may not have access to the business objects needed. Therefore, it is important to distinguish the vaults well.

For example, assume there are two vaults that contain information about Connecticut telephone customers. One is named “Waterbury Telephone Area” and the other is named “New Haven Telephone Area.” These names were chosen because they represent areas

defined within the regional phone directories. While it is clear where objects concerning New Haven and Waterbury telephone users are found, where are objects concerning users from the towns of Prospect, Bethany, and Beacon Falls? By including a Description clause, you can identify towns associated with each regional area:

Includes users from the towns of Bethany, Branford, East Haven, Hamden, Orange, North Brandon, North Haven, West Haven, and Woodbridge

For example, in the statements that follow, you can clearly identify the differences between the vaults being defined and where you might find the business objects related to telephone users from each town:

```
add vault "New Haven Telephone Area"
description "Includes users from the towns of
Bethany, Branford, East Haven, Hamden,
Orange, North Brandon, North Haven, West
Haven, and Woodbridge";
add vault "Waterbury Telephone Area"
description "Includes users from the towns of
Beacon Falls, Bethlehem, Middlebury,
Naugatuck, Prospect, Southbury, Thomaston,
Waterbury, Watertown, Wolcott, and Woodbury";
```

Icon Clause

Icons help users locate and recognize items by associating a special image with a vault. You can assign a special icon to the new vault or use the default icon. The default icon is used when in view-by-icon mode. Any special icon you assign is used when in view-by-image mode. When assigning a unique icon, you must use a GIF image file. Refer to [Icon Clause](#) in Chapter 1 for a complete description of the Icon clause.

GIF filenames should not include the @ sign, as that is used internally by eMatrix.

For example, you may want to use a company logo for the vault that contains objects related to doing business with that company. You could use a project logo for a project oriented vault, or an object icon (such as a tax form) for the vault containing those type objects.

Vault Types

There are three type of vaults: local, remote, and foreign. Most vaults are local. Remote vaults are used for loosely-coupled databases, which allow two entirely different eMatrix installations to share data. Foreign vaults are used with Adaplets™, which allow data from virtually any source to be modeled as eMatrix objects.

When defining a vault in MQL, you don't need to specify which type it is and there is no clause that allows you to specify the type. The system knows which type of vault you are defining by the parameters you specify for the vault. For local vaults, you define Oracle tablespaces using the Tablespace and Indexspace clauses. For remote vaults, you specify the server using the Server clause. For foreign vaults, you specify tablespaces, and Interface and Map fields (using the Interface and Map clauses). All these clauses are described in the following sections.

Tablespace Clause

The Tablespace clause is used to specify the Oracle tablespace in which the data tables for the vault are stored.

The names of the tablespaces and their associated storage are defined by the database administrator (DBA). This must be done prior to defining vaults. If you do not specify a tablespace name, the default data tablespace is used.

Refer to the *eMatrix System Manager Guide* for information on setting up tablespaces to optimize performance.

Indexspace Clause

The Indexspace clause is used to specify the Oracle tablespace in which the index and constraint information for the vault is stored.

The names of tablespaces and their associated storage are defined by the database administrator (DBA). This must be done prior to defining vaults. If you do not specify a tablespace name, the default index tablespace is used.

Refer to the *eMatrix System Manager Guide* for information on setting up tablespaces to optimize performance.

Server Clause

The Server clause of the Add Vault command is used to define the server for a remote vault. Most vaults will be Local; Remote vaults are vaults mastered in a Loosely-Coupled Database. Refer to *Sharing Data Between Federations* in the *System Manager Guide* for more information.

Use of the Server clause indicates that the vault is Remote.

Interface Clause

The Interface clause of the Add Vault command is used to define the full path name of the library for a foreign vault. The Interface should be specified as MATRIXHOME/api/mxff/mxff, with no extension. The .dll or shared library file is then used to access the vault, depending on whether the client is a Windows or a UNIX client.

Note that this field cannot be modified once the vault has been created. If changes are required, the vault must be deleted and then recreated. Deleting foreign vaults deletes only the eMatrix/database tables associated with it. The data from the foreign federation is left intact.

The interface can be used in more than one vault definition, but only by making a copy of it with a different name. For example, if the mxff interface will be used to link a second database with eMatrix (presumably with a different mapping), a copy of mxff should be created and renamed, and then referenced in the second foreign vault definition.

Use of the same interface (of the same name) by more than one vault will cause problems when accessing business objects.

File Clause

The File clause of the Add Vault command is used to define the map filename.

The scott.map file is placed in the MATRIXHOME/api/mxff/ directory when eMatrix is installed. This file should be opened in a text editor, and the contents copied into the schema map entry area. The map file can be specified with the File clause. For example:

```
add vault scott
  interface d:\matrix\api\mxff\mxff
  file d:\matrix\api\mxff\scott.map;
```

Map Clause

The Map clause of the Add Vault command is used to specify a string to define the schema map.

The schema map indicates which metadata in the foreign data maps to what types of data in eMatrix. It becomes part of the vault definition and is always referred to when accessing the data. The Map clause specifies the server, mode, and each table in the data source.

Refer to *Mapping the Data in the Building Adaplets to Other Federations: A Programmer's Guide* for more information.

Use of the Map clause indicates that the vault is Foreign.

Hidden Clause

You can specify that the new vault is “hidden” so that it does not appear in the Vault chooser in eMatrix. Users who are aware of the hidden vault’s existence can enter its name manually where appropriate. Hidden objects are accessible through MQL.

Property Clause

Integrators can assign ad hoc attributes, called Properties, to the vault. Properties allow associations to exist between administrative definitions that aren’t already associated. The property information can include a name, an arbitrary string value, and a reference to another administration object. The property name is always required. The value string and object reference are both optional. The property name can be reused for different object references, that is, the name joined with the object reference must be unique for any object that has properties.

```
add vault NAME
  property NAME [to ADMINTYPE NAME] [value STRING];
```

For additional information on properties, see *Overview of Administration Properties* in Chapter 24.

Modifying a Vault Definition

Modifying a Vault Definition

After a vault is defined, you can change the definition with the Modify Vault statement. This statement lets you add or remove defining clauses and change the value of clause arguments:

```
modify vault NAME [MOD_ITEM {MOD_ITEM}];
```

NAME is the name of the vault you want to modify.

MOD_ITEM is the type of modification you want to make. Each is specified in a Modify Vault clause, as listed in the following table. Note that you need to specify only fields to be modified.

Modify Vault Clause	Specifies that...
description VALUE	The current description, if any, changes to the value entered.
icon FILENAME	The image is changed to the new image in the file specified.
name NAME	The current vault name changes to the new name entered.
file MAPFILENAME	The name of the map file is changed.
map STRING	The map schema information is replaced with the new string.
hidden	The hidden option is changed to specify that the object is hidden.
nothidden	The hidden option is changed to specify that the object is not hidden.
property NAME [to ADMINTYPE NAME] [value STRING]	The named property is modified.
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.

Each modification clause is related to the clauses and arguments that define the vault. When modifying a vault, you first name the vault to change and then list the changes to make. For example, the following statement assigns new name and description to the vault called The Cleveland Project.

```
modify vault "The Cleveland Project"
    name "The San Diego Project"
    description "Includes data for San Diego area."
    file c:\ematrix\scott.map;
```

Clearing and Maintaining Vaults

Clearing Vaults

The Clear Vault statement is used to delete all business objects and associations in a vault:

```
clear vault NAME [ ,NAME ];
```

NAME is the name of the vault you want to clear.

Once a Vault has been established and used, you should NOT use the Clear Vault statement. In addition, you should not use it when users are online.

Indexing Vaults

The Index command should be used periodically, after modifications and deletions, to clean up the database indices.

```
index vault NAME [table TABLE_NAME] [indexspace TABLESPACE_NAME];
```

Re-indexing vaults can improve find performance whether or not transaction boundaries have been used in the data loading process. If data is loaded from a sequentially sorted data file, the resulting index will be less than optimal. Re-indexing *randomizes* the index, making find performance noticeably better. Indexing a vault in this manner rebuilds the system indices that must be present for locating objects by name, type, and owner, as well as other indexed fields.

To show the SQL commands for a particular index vault command, without actually changing the indices, use the validate index vault command as follows::

```
validate index vault NAME [table TABLE_NAME] [indexspace TABLESPACE_NAME];
```

This is helpful to use on very large databases, where indexing a vault may take many hours. The validate output shows the SQL commands that need to be run. You could then manually run the commands in order, to make progress with minimal disruption.

Each clause is described below.

table clause

Include the table clause to indicate which database tables should be re-indexed. Only those columns that have indexing defined will be re-indexed. You should include up to and including the “_” in a table name, since what follows is specific to the the vault specified. For example:

```
index vault "Engineering-1" table lxbo_;
```

This command might generate and execute SQL similar to:

```
alter index lxBO_abbe6b7a_lxOid_Index rebuild;
alter index lxBO_abbe6b7a_lxName_Index rebuild;
alter index lxBO_abbe6b7a_lxOwner_Index rebuild;
alter index lxBO_abbe6b7a_lxPolicy_Index rebuild;
```

The Engineering-1 vault is associated with the abbe6b7a table.

indexspace clause

Use the `indexspace` clause to specify an alternate database tablespace to use for processing this command. For example:

```
index vault "Engineering-1" table lxbo_ indexspace USER_DATA;
```

This SQL generated is as follows:

```
alter index lxBO_abbe6b7a_lx0id_Index rebuild tablespace USER_DATA;
alter index lxBO_abbe6b7a_lxName_Index rebuild tablespace USER_DATA;
alter index lxBO_abbe6b7a_lxOwner_Index rebuild tablespace USER_DATA;
alter index lxBO_abbe6b7a_lxPolicy_Index rebuild tablespace USER_DATA;
alter index lxBO_abbe6b7a_lxState_Index rebuild tablespace USER_DATA;
```

This command adds indices to all columns of `lxBO_` table (of the vault `Engineering-1`) that have indexing defined, and the command would use tablespace `USER_DATA` to hold the index data.

Fixing Fragmented Vaults

As objects are deleted from a vault, storage gaps will occur in the vault database file. These gaps represent wasted disk space and can cause an increase in access time. MQL provides the `tidy vault` statement to fix fragmentations in the database file of the vault.

```
tidy vault NAME [commit N];
```

`NAME` is the name of the vault you want to fix. You can specify the `ADMINISTRATION` vault to remove unused records of deleted administration objects.

When this statement is executed, eMatrix consolidates the fragmented database file. It deletes rows in the database tables that are marked for deletion.

commit N clause

Include the `commit N` clause when tidying large vaults. The number `N` that follows specifies that the command should commit the database transaction after this many objects have been tidied. The default is 1000. For example:

```
tidy vault "Engineering" commit 200
```

Since the `tidy vault` statement creates a temporary database file in the location of the vault while it is running, it requires free disk space equal to the size of the original database file.

Updating Sets With Change Vault

When an object's vault is changed, by default the following occurs behind the scenes:

- The original business object is cloned in the new vault with all business object data except the related set data
- The original business object is deleted from the “old” vault.

When a business object is deleted, it also gets removed from any sets to which it belongs. This includes both user-defined sets and sets defined internally. IconMail messages and the objects they contain are organized as members of an internal set. So when the object's

vault is changed, it is not only removed from its sets, but it is also removed from all IconMail messages that include it. In many cases the messages alone, without the objects, are meaningless.

To address this issue, the change vault command includes the following additional functionality:

- Add the object clone from the new vault to all IconMail messages and user sets in which the original object was included.

The additional functionality may affect the performance of the change vault operation if the object belongs to many sets and/or IconMails. For this reason, the default functionality has not been changed, but business administrators can execute the following MQL command to enable/disable this functionality for system-wide use:

```
set system changelattice update set | on | off |;
```

For example, to turn the command on for all users, use:

```
set system changelattice update set on;
```

Once this command has been run, when users change an object's vault via any application (that is, eMatrix desktop applications, eMatrixApplet, MatrixOne's Value Chain Portfolio applications, or custom ADK applications), all IconMails that reference the object are fixed.

You can also fix IconMail at the time the change vault is performed (in MQL only). For example:

```
modify bus Assembly R123 A vault Engineering update set;
```

Printing a Vault Definition

The Print Vault statement prints the vault definition to the screen allowing you to view it. When a Print statement is entered, MQL displays the various clauses that make up the vault definition and their current values.

```
print vault NAME;
```

If the NAME contains embedded spaces, use quotation marks.

A print vault statement lists only objects that have been updated (those objects for which there is a eMatrix/database table entry).

Deleting a Vault

If a vault is no longer required, you can delete it with the Delete Vault statement:

```
delete vault NAME;
```

NAME is the name of the vault to be deleted.

Only empty vaults can be deleted. To remove all objects from a vault, use the `clear vault` command. See [Clearing Vaults](#).

When this statement is processed, eMatrix searches the list of existing vaults. If the name is found and the vault contains no business objects, the vault is deleted. If the name is not found, an error message is displayed. If you attempt to delete a vault that contains business objects, an error message is displayed.

For example, to delete the vault named “1965 Bank Loans,” enter:

```
delete vault "1965 Bank Loans";
```

After this statement is processed, the vault is deleted and you receive an MQL prompt for another statement.

5

Working With Stores

Store Defined	124
Types of File Stores.....	125
Ingested Files	125
Captured Files	125
Tracked Files	126
Defining a Store	127
Description Clause.....	128
Icon Clause.....	128
Type Clause.....	129
Filename Hashed Clause	129
Lock Clause	131
Path Clause	131
Host Clause	131
Protocol and Port Clauses.....	132
Permission Clause.....	133
User and Password Clauses	134
URL Clause	134
Tablespace Clause	135
Indexspace Clause	135
Hidden Clause	135
Property Clause	135
Enabling Secure FTP for a Captured Store	136
Modifying Store Definitions	138
Modifying a File Store	138
Maintaining a File Store	140
Tidy Store Statement.....	140
Inventory Store Statement.....	140
Rehash Store Statement	141
Validate Store Statement.....	142
Monitoring Disk Space.....	142
Deleting a Store	144
Purging Files.....	144

Store Defined

A *store* is a storage location for checked-in application files. All files checked in and used by eMatrix are contained in a file store. These files can contain any information and be associated with any variety of business objects. A file store simply defines a place where you can find the file. You can define file stores for CAD drawings, documentation files, problem reports, and so on.

A store is used to divide the database for improved performance. The amount of control eMatrix has over the physical storage, retrieval, and security of a file is dependent on the type of store used. A file store provides:

- Information on the physical storage location of a file checked into eMatrix. Stores, like vaults, should be strategically placed within the network topology.
- Access to that information. Consider disk storage requirements—the store contains more data than typically found within the vault.
- Three optional storage methods. Three different types of stores (ingested, captured, and tracked) provide varying degrees of file security, performance, and access.
- Access to files using NFS or FTP. Captured stores can be configured to use NFS or FTP for file access. In addition, this type of store can have alternate locations where the data is replicated, improving access to it from distant sites. Refer to [Captured Store Replication](#) in Chapter 6.
- Full text search. A captured store can be configured to provide full text search capabilities for the files it contains. Indexing software is also required.

You must be a System Administrator to access stores.

If the store is to be physically located on a PC and accessed through a network drive, the store host name must be set to LOCALHOST.

Multiple file stores are possible at different locations. For example, two or more file stores could contain CAD drawings. These stores might be located on the same host or on different hosts. An object's policy determines the store that is used for its files.

A lock feature enables you to lock a store from the user for all write activities. Business objects with a policy using a locked store cannot have files checked in (written), but files can be checked out (read). This is useful when a store becomes obsolete. Stores are unlocked by default.

For information on replacing a store with a new store, see [Implications of Changing Stores](#) in Chapter 6.

Types of File Stores

There are three types of file stores:

- Ingested
- Captured
- Tracked

Each type specifies the amount of control and knowledge eMatrix has over the files. As such, each type has different parameters associated with it.

Stores should use actual host names and paths, not mounted directories. (However, you may want to use mounted directories for tracked stores. This will make the files seem local and behave better for launching open/edit/etc.)

Ingested Files

An *ingested file store* contains *ingested files*, which are completely controlled by eMatrix. The information within the files is subject to eMatrix access control. This results in the fastest processing times and the simplest file maintenance. Once a file is ingested, it can be retrieved only using eMatrix and cannot be accessed using any of the system file utilities.

Captured Files

A *captured file store* contains *captured files*. A captured store offers a bit more flexibility than an ingested store in regard to system control while still taking advantage of eMatrix's file and access control. Captured files are maintained by eMatrix and are subject to the access rules defined in the policy that governs the file associated with the business object. The primary means of accessing the file is from eMatrix although it is possible to access it from the file system.

Captured stores provide some features that are not available to other types of stores. FTP can be used in addition to NFS or UNC paths as the method of accessing files from remote systems. In addition, full text search of file content can be configured only for captured stores.

Files should be accessed through eMatrix

Files in captured stores should not be manipulated or altered outside of eMatrix (for example, through the operating system). If a file that is being checked out is a different size than when it was checked in, the following warning message displays:

"File size has changed from XXXX to YYYY since it was checked in. File may be damaged."

Filename hashing

Captured stores can use file name *hashing*, which is the ability to scramble the file name. When file name hashing is on (the default), captured stores generate hashed names for checked in files based on a random number generator and timestamp. If a name collision occurs, it will retry with a new hashname up to 100 tries, then return an error. Since the

files for captured stores are physically stored on disk, the names are hashed to be recognized by eMatrix only.

When file name hashing is off, the file names appear in the protected captured directory with original file names. Unhashed file names *collide* in a store more often than when eMatrix generates unique hashed file names for each checked-in file. Since two physical files of the same name cannot reside in a single directory, eMatrix will scramble the name of one copy whenever a collision would occur.

Tracked Files

A *tracked file store* contains *tracked files*, which provide the least amount of eMatrix control. eMatrix maintains information about the file but does not control the physical file itself—the naming, maintenance, and general access is controlled external to eMatrix. A tracked file maintains the maximum amount of external control while allowing some access from within eMatrix.

When a file is placed in a tracked file store, it becomes accessible to other eMatrix users based upon the policy assigned to the business object associated with the file. For example, if the business object's policy allows group access, any file associated with that business object can be accessed by members of the defined eMatrix group. This is true even if no external (outside of eMatrix) access is allowed by the file owner.

NFS must be installed on all the machines involved.

When using tracked files, direct physical access of a file via a system file utility can disrupt eMatrix access. If you rename, alter, or move any physical file, you will have to check in the file again to maintain eMatrix access. If you don't, when file access is attempted via eMatrix, errors will occur

Tracked File Usage

Tracked file stores are useful when you have a private workstation that contains files not normally accessible to other eMatrix users. In this case, the created files are not governed by a policy and public access to the files is restricted. If you have a file that you want to make public to the eMatrix users only, you can do so by checking it into eMatrix in a tracked file store.

This enables you to maintain complete control over the physical storage of the file while allowing others to access it from within eMatrix. Users outside of eMatrix are prohibited from accessing the file because it is on your workstation, while users within eMatrix can access it according to the policy assigned when the business object containing the file was created.

Tracked files are also useful when you have extremely large files that you do not want to move. It is cumbersome to move and easily manage large files. Rather than moving a file to the user (to a local storage area for captured files), it is easier to direct the eMatrix user to where the file physically resides (stored as a tracked file).

Since tracked files exist in the directory structure of the file system and are not copied into eMatrix, it does not make sense to check a tracked file out to the machine where the file resides. If directory check-out is attempted into a directory of the same name, the function will fail.

Defining a Store

An eMatrix file store is defined with the Add Store Statement:

```
add store NAME [ADD_ITEM {ADD_ITEM}];
```

NAME is the name of the store you are defining. All stores must have a name unique in the database. The name can be up to 127 characters long and can contain spaces. You should assign a name that has meaning to both you and the user. For example, each of the following is a valid file store name:

Electronic CAD Drawings
X29 Development
Income Taxes

ADD_ITEM is an Add Store clause that provides more information about the store you are creating. The only required clause for store creation is the type clause. Other clauses can help to further define the store. The Add Store clauses are:

description VALUE
icon FILENAME
type captured ingested tracked
filename [not]hashed
[un]lock
path PATH_NAME
host HOST_NAME
protocol PROTOCOL_NAME
port PORT_NUMBER
permission OWNER_ACCESS [,GROUP_ACCESS [,WORLD_ACCESS]]
user USER
password PASSWORD
url VALUE
tablespace SPACE
indexspace SPACE
[! not]hidden
property NAME [to ADMINTYPE NAME] [value STRING]

The Path clause is required only for captured or ingested files. Although all other clauses are optional, the following defaults are assumed:

Clause	Default
filename	hashed
host	localhost
indexspace	default index tablespace (as defined by the DBA)
unlock	
path	MATRIXHOME
permission	rw, rw, rw
tablespace	default data tablespace (as defined by the DBA)

Each clause and the arguments they use are discussed in the sections that follow.

Description Clause

The Description clause of the Add Store statement can provide general information for both you and the Business Administrator about the purpose of the store. It can also help point out subtle differences between stores to the user.

For example, you might have two file stores that contain captured CAD drawing files. One store might be used for a particular project or group while the other is used for another project or group. Having a file store for each group can help with performance. Different groups or projects may have different memory allocation or cleanup requirements. You can determine the requirements for each and then define them accordingly.

A Description clause should clearly indicate the differences between each store. For example, you can identify the differences between the file stores and the kinds of files that might be stored in each example:

```
add store "Electronic CAD Drawings"
description "Stores captured electronic CAD drawing files"
type captured;

add store "Vellum CAD Drawings"
description "Stores vault locations of hardcopy CAD drawings"
type tracked;

add store "X29 Development"
description "Stores ingested files related to development of X29 solar
vehicle"
type ingested;
```

Icon Clause

The Icon clause of the Add Store statement associates an image with a file store. Icons help users locate and recognize items. You can assign a special icon to the new store or use the default icon. The default icon is used when in view-by-icon mode. Any special icon you assign is used when in view-by-image mode. When assigning a unique icon, you must use a GIF image file.

For example, you may want to use a project icon for the file store used by that project or an icon that represents the types of files found in the file store (a drawing icon for drawing

files, a paper and pen icon for text files, and so on.). Refer to *Icon Clause* in Chapter 1 for a complete description of the Icon clause.

GIF file names should not include the @ sign, as that is used internally by eMatrix.

Type Clause

The Type clause of the Add Store statement identifies how the files are stored and accessed. You can specify one of the three file storage types: ingested, captured, or tracked (as described above).

eMatrix is not responsible for tracked files. This can make backups much more difficult—the System Administrator must ensure that all physical files assigned to the tracked file store are backed up.

Once you decide how to store and control your files, you can specify the store type by adding a Type clause to your Add Store statement. All file store definitions must have a Type clause to be usable. The following example creates a store to hold captured files:

```
add store Drawings
    description "Storage for electronic drawings"
    host RELIABLE
    path ${MATRIXHOME}/Drawings
    type captured;
```

Filename Hashed Clause

The Filename Hashed clause of the Add Store statement applies when using captured stores only. It enables you to scramble the file name. When file name hashing is on (hashed), captured stores generate hashed names for checked in files. Since the files for captured stores are physically stored on disk, the names are hashed to be recognized by eMatrix only. For example:

```
add store "Electronic CAD Drawings"
    type captured
    filename hashed;
```

When file name hashing is off (nothashed), the file names appear in the protected directory as the original file names. For example:

```
add store "Electronic CAD Drawings"
    type captured
    filename nothashed;
```

Unhashed file names will *collide* in a store more often than if eMatrix generates unique hashed file names for each checked-in file.

Algorithm for hashed files

Hashnames are based on a random number generator and timestamp.

If a collision occurs, eMatrix retries with a new hashed filename up to 100 tries. If it cannot find a unique name within 100 retries, the transaction aborts and the message “Unique file cannot be generated in captured store” displays. In this way eMatrix guarantees uniqueness within a store/location.

A file copied from a store to a location goes through the same naming captured file algorithm. In theory its generated name could happen to be the same as in the store, though extremely unlikely.

Algorithm for non-hashed store

In the case where a file is copied to a store that already contains a file of the same name, eMatrix automatically generates a hashed name in order to prevent a filename collision.

A file copied from a store to a location tries to retain the same non-hashed name. If there is a collision, eMatrix generates a hash name for it and displays a message that hash file name was generated.

File Name Hashing and Full Text Search

Some query engines, including Microsoft Index server, use the file extension to determine file type (such as “.doc” for Microsoft Word). For this reason, there are two hashing algorithms that eMatrix may use. When generating a hashed file name for a store or location that contains a URL definition, the file extension is preserved. If no URL is specified, the extension is hashed as well. So, when hashing a file called “Procedure.doc,” the hashed name would look something like “12345678.doc” if a URL is specified and “12345678.abc” if no URL is specified.

If you are using Microsoft Index Server or another search engine that needs to know the file extension, and you are indexing a set of documents that have been checked in prior to setting up the full text search capability, you must re-generate the hashed names. The [Rehash Store Statement](#) is available for this purpose. This command simply iterates through each file in a captured store and regenerates its file name based on the definition of the store.

Microsoft Index Server (and perhaps other index servers, too) requires hashing for another reason as well. This query engine converts all file names to lower case. If files containing upper case characters have been checked in, a query for those files will fail unless the file names are hashed (which guarantees lower case names). Note that hashing does not change the case of the file *extension*, so it is important to be sure that the extension is not capitalized before checking in a file.

Full Text Search Limitation

When using eMatrix full text search, spaces may be translated to hexadecimal (%20). For example, when searching for a file named “Data Model.doc” some search engines would actually look for “Data%20Model.doc”. When this file was found, it would then not be able to identify the business object that is associated with it, since the filename eMatrix knows about contains the space. Files that do not include spaces in their names are found as expected.

The only workaround is not to check in or search for files with spaces in their names.

Lock Clause

The Lock clause locks a store from the user for all write activities. Business objects with a policy using a locked store cannot have files checked in (written), but files can be checked out (read). This is useful when backups are being made or when a store is full.

```
add store "Vellum CAD Drawings"  
locked  
type ingested;
```

The default is `unlocked`.

```
add store "Vellum CAD Drawings"  
unlocked;  
type ingested;
```

Path Clause

The Path clause of the Add Store statement identifies where the file store is to be placed on the host. The Path clause is required for captured stores. Paths should not include the @ sign, as that is used internally by eMatrix.

Stores should use actual host names and paths, not mounted directories. (However, you may want to use mounted directories for tracked stores. This makes the files seem local and behave better for launching open/edit/etc.) The location of captured stores must have the path exported to all users that need access to the files, or captured stores can be accessed via FTP.

A captured store will create a directory for the captured files. If the captured store will be accessed via FTP, the Path can be relative to the FTP username login, but not necessarily the root directory of the FTP server system. For example, if the Path is specified as Drawings, and doing an FTP login puts you on the host machine in the `usr/matrix` directory, the FTP store would be physically located in `usr/matrix/Drawings`. An absolute path can also be entered; however, the parent directory must already exist. For example, if the FTP Store path is specified as `"/stores/store1/"` and the `"/stores"` directory does not exist on the target host, the store will not be created.

The FTP server must use UNIX directory listing option, even on NT. DOS directory style is not currently supported.

An ingested store does not create a directory. It creates a file in an existing directory into which it will ingest the checked in files.

The following Add Store statement creates a file store located in the `MATRIXHOME` Training subdirectory on the Reliable host:

```
add store Training  
description "Storage for training information"  
host Reliable  
path ${MATRIXHOME}/Training  
type captured;
```

Host Clause

The Host clause of the Add Store statement identifies the host system that is to contain the file store being defined. If you want to use the host on which you are currently working, you do not need to include this statement. If a host is not specified, the current host is

assumed and assigned. Host names should not include the @ sign, as that is used internally by eMatrix.

If the store is to be physically located on a PC and accessed through a network drive, the store host name must be set to localhost.

A host is not necessary for a tracked store because each tracked file would contain its own path/host specification.

File stores can be created and can exist across networks. Depending on who uses a file store, you might install it on a system local to its users. That speeds up access and avoids placing additional loads on network communications.

File stores that are frequently used by all nodes within a network should be contained on a centrally located node. If a file store is used extensively by one system, you may want to place the store on that system to improve access time and communications requirements.

For example, the following statement defines a store (Video) that resides on the system called Reliable in a directory defined by the environment variable MATRIXHOME:

```
add store Video
  description "Storage for video data"
  host Reliable
  path ${MATRIXHOME}
  type captured;
```

Protocol and Port Clauses

When creating a captured store, you can include the parameters protocol and port (the same is true for locations).

```
protocol PROTOCOL_NAME
```

PROTOCOL_NAME can be either `ftp`, `ftps`, or `nfs`.

You can enter `ftps` as a protocol value only if secure FTP is enabled on your system. See [Enabling Secure FTP for a Captured Store](#).

Each protocol has a default port that is used if not specified in the store definition. Include the port clause to specify a port other than the default.

```
port PORT_NUMBER
```

In support of this change, as of version 9, file names in error messages no longer use the eMatrix-specific format:

`/DIR/NAME@SERVER`

but will now use a Web-like URL:

`PROTOCOL:/LOCATIONHOSTNAME:PORT/LOCATIONPATH/FILENAME`

For example:

```
add store NFSStore type captured host Reliable
  path ${MATRIXHOME}/NFSStore protocol NFS;
```

If a store/location does not have a protocol specified, eMatrix looks at other object attributes to determine which protocol was likely intended.

- If the host is ‘localhost’ (or empty), the protocol used will be ‘file’ (local file system).
- If the host is *not* ‘localhost’ and a username/password was given, then the store will use `ftp`.

- If the host is *not* ‘localhost’ and there is *no* username/password, the store will use nfs.

Note these checks eliminate the need to add protocol/port parameters to store/locations added prior to version 9.

Permission Clause

For captured store only, the Permission clause specifies who will have the ability to read, write, or execute a database being stored. There are three categories of users:

- Owner
- Group
- World

These categories are assigned and controlled by your operating system. They are not the same as an Owner or Group within eMatrix—instead, Owner and Group are categories that define access (as described below).

The Permission clause uses the following syntax:

```
permission OWNER_ACCESS [,GROUP_ACCESS [,WORLD_ACCESS]]
```

For each category, you specify the type of permission:

r	Read access permission
w	Write access permission
x	Execute access permission

The default is rw, rw, rw which indicates Owner read/write access, Group read/write access, and World read/write access.

If you are specifying permissions, you must include at least the Owner Access. If you choose to define Group Access or World Access, the Owner Access or Owner and Group Access must precede it, respectively.

For example, each of following definitions include a valid Permission clause:

```
add store "Accounting Files"
    permission rw;

add store "Personnel Files"
    permission rw, r;

add store "Library Files"
    permission rw, rw, r;
```

In the first example, only the owner of the files has read and write access to them. In the second example, the file owner has read and write access while the group has only read access. In the third example, both the owner and group may read and write, and everyone else has read access.

When setting the values for the permission, you need to know how ownership of the database files are assigned from within the operating system. If they were assigned to a single owner, you will want to set the Owner values for full access. If they were assigned to a group, the group should have full access.

User and Password Clauses

The User and Password clauses are used to specify the FTP username and password. When moving files to/from a captured store, the FTP username defines FTP account users for the store. If FTP information is not provided, the system uses NFS to move files. The FTP password provides access to the FTP account.

Before an FTP store can be used, the store's host system must be configured to act as an FTP server, and the FTP Username and Password specified here must be established. eMatrix has FTP client functionality built in so no additional software or configuration is necessary on the eMatrix workstations. See your operating system documentation for more information on installing and configuring FTP.

URL Clause

To implement URL full text searching, a URL must be defined in captured Stores. The URL is in the form of a CGI-bin request. It is assumed that the CGI-bin program and the indexing engine used by the URL is installed and administered independently of eMatrix.

Each Store should specify the following URL:

`http://${HOST}/matrix/mxis.asp?ct=${LOCATION}&qu=${QUERY}&mh=${LIMIT}`

During query evaluation, these variables are expanded as follows:

- LOCATION is expanded to the name of the eMatrix store object
- HOST is expanded to contain the host of the eMatrix store object
- QUERY is expanded to contain the search string
- LIMIT is expanded to contain the contents of the MX_FULLTEXT_LIMIT variable in each users' initialization file

If you are using FTP stores, the paths will vary from the paths used to create the directories for the index server catalog.

Matrix.ini Settings

Some search engines may need more information than provided with the basic URL shown above. Additional common parameters to be used with the full text search integration can be set in the Matrix.ini file and added to the URL specified in the store. For example, MODE, LIMIT, and MINSCORE settings can be passed to a search engine by setting the following variables in the .ini file:

- MX_FULL_TEXT_MODE
- MX_FULL_TEXT_LIMIT
- MX_FULL_TEXT_MINSCORE

The URL should be modified using variable syntax. For example, to add the MINSCORE setting use:

`http://${HOST}/matrix/mxis.asp?ct=${LOCATION}&qu=${QUERY}&mh=${LIMIT}&score=${MINSCORE}`

The possible values for these variables depend on the search engine used. However, many have the concepts of a simple versus a complex *MODE*, a *LIMIT* to the number of files to return, and a *MINSCORE* value which the file must meet. Also, up to three additional

parameters can be passed to the search engine using the ARG1, ARG2, and ARG3 variables in the .ini file.

If you set any of these variables in the .ini file and add them to the URL of the store, eMatrix will provide them to the search engine, but the search engine will then be responsible for interpreting the settings.

Refer to *Enabling Text Searches on Captured Stores* in the *eMatrix System Manager Guide* for more information.

Tablespace Clause

The Tablespace clause is used to specify the Oracle tablespace in which the data tables for the ingested store reside.

The names of the tablespaces and their associated storage are defined by the database administrator (DBA). This must be done prior to defining vaults. If you do not specify a tablespace name, the default data tablespace is used.

Refer to the *eMatrix System Manager Guide* for information on setting up tablespaces to optimize performance.

Indexspace Clause

The Indexspace clause is used to specify the Oracle tablespace in which the index and constraint information for the ingested store resides.

The names of tablespaces and their associated storage are defined by the database administrator (DBA). This must be done prior to defining vaults. If you do not specify a tablespace name, the default index tablespace is used.

Refer to the *eMatrix System Manager Guide* for information on setting up tablespaces to optimize performance.

Hidden Clause

You can specify that the new store is “hidden” so that it does not appear in the Store chooser in eMatrix. Users who are aware of the hidden store’s existence can enter its name manually where appropriate. Hidden objects are accessible through MQL.

Property Clause

Integrators can assign ad hoc attributes, called Properties, to the store. Properties allow associations to exist between administrative definitions that aren’t already associated. The property information can include a name, an arbitrary string value, and a reference to another administration object. The property name is always required. The value string and object reference are both optional. The property name can be reused for different object references, that is, the name joined with the object reference must be unique for any object that has properties.

```
add store NAME  
    property NAME [to ADMINTYPE NAME] [value STRING];
```

For additional information on properties, see *Overview of Administration Properties* in Chapter 24.

Enabling Secure FTP for a Captured Store

Secure FTP (FTPS) builds on the standard FTP protocol by adding strong encryption and authentication methods, providing secure password management and encrypted file content between the client and the server. eMatrix can be used to communicate via secure FTP using the secure FTP server provided by the Stanford Secure Remote Password (SRP) Authentication Project.

MatrixOne has qualified the secure FTP support using version 1.7.2 of the SRP server on the HP platform only.

The main tasks for configuring eMatrix to support secure FTP are:

1. Download and install the OpenSSL toolkit.
2. Download and set up the SRP version 1.7.2 server.
3. Configure eMatrix as a secure FTP client by defining a secure FTP store.

To set up OpenSSL and the SRP server

1. Download the OpenSSL toolkit from <http://www.openssl.org/>. Install it to the machine on which you will set up the SRP server.
2. Download SRP version 1.7.2 from <http://srp.stanford.edu>. From the SRP home page, click **Download it here**, then click **srp-1.7.2.tar.gz**.
3. Set up the SRP server on the desired platform by following the instructions in the documents that come with the server installation.

There may be a significant performance impact if you choose to encrypt file content, which is the default configuration. To turn off data encryption but still have username/password hashed, you can use the NO_ENCRYPTION option. This option is a compile time flag on the server.

Building the FTP server requires the compilers/linkers appropriate for the chosen platform, as specified in the SRP Project documentation.

4. Install the resulting binary server/daemon and password following the instructions in the provided documentation.
5. Qualify the build/installation independently from eMatrix using any FTP client. The SRP Project site provides a secure FTP client you can use to test the server. Also confirm that the system on which the SRP is installed can be connected to using any of the other tools your company may require to access the system, such as ordinary FTP, telnet, mount, automount (from UNIX systems), mapped drives (Windows), remote login, remote shell, xwindows (unix) and emulators (Windows).

Don't attempt to connect using eMatrix until you are sure the SRP server works correctly.

To configure eMatrix to support secure FTP

Make sure the SRP server works with a standard FTP client before attempting to use the eMatrix client.

- 1.** Using System Manager (or MQL), define a captured store and specify **ftps** as the Protocol for the FTP store.
Based on this value, eMatrix will load a shared library called mxftps.
- 2.** In all other fields for the store—such as port, host, path, FTP username, and FTP password—use the same values that you would for any FTP store.

See [*Defining a Store*](#) for details.

Modifying Store Definitions

Modifying a File Store

After a file store is defined, you can change the definition with the Modify Store statement. This statement lets you add or remove defining clauses and change the value of clause arguments:

```
modify store NAME [MOD_ITEM {MOD_ITEM}];
```

NAME is the name of the store you want to modify.

MOD_ITEM is the type of modification you want to make. Each is specified in a Modify Store clause, as listed in the following table. Note that you need to specify only fields to be modified.

Modify Store Clause	Specifies that...
add location LOCATION_NAME	The named location is added to the store.
remove location LOCATION_NAME	The named location is removed from the store.
description VALUE	The current description, if any, is changed to the value entered.
icon FILENAME	The image is changed to the new image in the file specified.
name NAME	The current file store name is changed to that of the new name entered.
filename [not]hashed	The file name is either encoded (hashed) or not (nohashed).
path PATH_NAME	The name of path to the file store is changed to the value entered. Note: If you change the path of a store, the files are not accessible. When you change the path, the system assumes you are also going to move the files.
protocol PROTOCOL_NAME	The protocol is changed to the value specified.
port PORT_NUMBER	The port is changed to the number specified.
host HOST_NAME	The host containing the file store is changed to the host named.
[un]lock	The store is either locked from the user for all write activities (lock) or available (unlock).
permission OWNER_ACCESS [,GROUP_ACCESS [,WORLD_ACCESS]]	Access permission is set to the values entered. The order in which the permission values are entered determines that they are assigned to the Owner, Group, or World category of users.
user USER	The current FTP username is changed to the new name entered.
password PASSWORD	The current FTP password is changed to the new one entered.

Modify Store Clause	Specifies that...
url VALUE	The current URL is changed to the new one entered.
hidden	The hidden option is changed to specify that the object is hidden.
nohidden	The hidden option is changed to specify that the object is not hidden.
property NAME [to ADMINTYPE NAME] [value STRING]	The named property is modified.
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.

As you can see, each modification clause is related to the clauses and arguments that define the file store. The only modifications that you cannot make to a file store definition is a change in the store type or the tablespaces used. To change the store type, you must create a new file store. Once a store type is assigned, it remains in place as long as the file store exists. When you modify a store, you first name the store and then list the modifications. For example, the following statement changes the name of the Drawings store and the path:

```
modify store Drawings
  name "CAD Drawings"
  path ${MATRIXHOME}/CADDrawings;
```

When this statement is executed, the name of the store changes to “CAD Drawings” and the path is changed to \${MATRIXHOME} / CADDrawings.

Maintaining a File Store

MQL provides two statements that help you maintain file stores: the Tidy Store statement and Inventory Store Statement.

Tidy Store Statement

As files are checked in and, consequently, older versions are deleted, the ingested store file may become fragmented. The Tidy Store statement defragments the database file.

The Tidy Store statement can also be used in a captured store situation to clean up obsolete copies of files that might exist in a replicated environment.

```
tidy store NAME;
```

NAME is the name of the file store you want to tidy.

When this statement is executed, eMatrix removes obsolete copies of files and consolidates the fragmented database file.

Since the tidy store statement creates a temporary database file in the location of the store, it requires free disk space equal to the size of the original database file.

Inventory Store Statement

System Administrators can list the file contents of a store by using the Inventory Store statement. Additional clauses provide the ability to specify a subset of locations to inventory. Memory usage remains at a low fixed amount regardless of the size of the store. The syntax is:

```
inventory store NAME [location LOCATION_TARGET{,LOCATION_TARGET}] [store];
```

NAME is the name of the file store.

LOCATION_TARGET is one or more location names that are associated with the specified store. Locations are separated by a comma but no space.

It is highly recommended that inventories of all stores and locations are performed nightly as part of the backup procedure.

If file names are hashed, a list is presented containing a mapping of the hashed names to the original names.

If MQL is running in verbose mode, the output includes:

- The full path of file origination
- The date and time of creation (if the file name is hashed)
- The format of the file and the owning business object.

You can also include the keyword store after the storename to list files stored at the store itself. For example, the command:

```
inventory store Assemblies store location Dallas,London;
```

might result in the following output:

```
store Assemblies file original
```

C:\Designs\widget\SHEET1.PRT captured /matrix/prod/stores/
Assemblies/31b49a45.784 on Tue Apr 4, 1994 4:19:17 PM format
Cadra of business object Assembly 12345 A

The creation date and time for hashed file names correlates to when the database received control of the file and hashed the name. Keep in mind that there are a number of ways a file can get into eMatrix: through checkin, import, cloning, or revisioning.

If the output indicates that files have been orphaned, contact MatrixOne Customer Service for help in diagnosing the error.

Rehash Store Statement

The Rehash Store statement iterates through each file in a captured store and regenerates its file name based on the store's definition. This command is particularly useful under the following conditions:

- A store is changed from hashed to non-hashed and the administrator wants to restore the original names of the files.
- A store is changed from non-hashed to hashed and the administrator wants to hash all the names in the store.
- A URL is added to a store that changes the name hashing algorithm. In this case, the administrator wants to regenerate the hashed names to be compatible with full-text-search (that is, the extension is not hashed).
- When hashed files are created in a non-hashed store when importing a business object, and the administrator wants to quickly change all files in the store back to non-hashed files.

The command syntax is:

```
rehash store NAME;
```

File Hashing Changes for Full Text Search

Some query engines, including Microsoft Index server, use the file extension to determine file type (such as ".doc" for Microsoft Word). For this reason, when generating a hashed file name for a store or location that contains a URL definition, the file extension is preserved. So, when hashing something like "Procedure.doc," the hashed name would look something like "12345678.doc" instead of "12345678.abc."

If you are using Microsoft Index Server or another search engine that needs to know the file extension and you are indexing a set of documents that have been checked in prior to version 7, you must re-generate the hashed names. The `rehash store` command simply iterates through each file in a captured store and regenerates its file name based on the definition of the store.

Microsoft Index Server (and perhaps other index servers, too) requires hashing for another reason as well. This query engine converts all file names to lower case. If file names are not hashed (thereby guaranteeing lower case names), some files will not be found when the names of checked in files are not all lower case.

Note that hashing does not change the case of the extension, so it is important to be sure that the extension is not capitalized before checking in a file.

Validate Store Statement

The Validate Store statement can be used to identify files in a captured store that are not associated with any business object in the eMatrix database. The command requires an input file that lists by name all files in the store's directory. This input file can be created from the captured store directory using the following command on UNIX:

```
ls > FILENAME
```

or from a DOS prompt on Windows:

```
dir > FILENAME
```

Once the input file is created, System Administrators can use the following command:

```
validate store NAME file FILENAME;
```

Where:

NAME is the name of a captured store.

FILENAME is an input file that lists all files in a captured store directory.

The validate store command creates a file called FILENAME.out (the same name as was used for the input filename, except with a file extension of .out), which lists files that are not associated with business objects. You can then cleanup the store's directory by deleting these files.

Monitoring Disk Space

As a System Administrator, you should monitor the disk space available to your stores carefully. If disk space is running low, you could simply change the path or host of the store, but then you would have to move all checked-in files to the new directory, or users would receive errors when trying to access the files. A better solution is to change the store that the policies use.

To replace an existing store

1. Use the following MQL command:

```
modify store STORE_NAME lock;
```

2. Create a new store with a host and path that has ample disk space.

3. Determine which policies use the store that needs to be replaced.

- a) In MQL, run the following command:

```
list policy * select name store dump | ;
```

The results will look something like:

```
CMM|Object  
Model|Product  
Critical Report|Submission  
Incident|Submission2  
Sequencer|Object  
Media|Product  
Task|Object  
Marketing Documents|Document  
Kit|Product  
...
```

- b) Copy the list and paste it into a text editor.

- c) Create a table and sort by the policy column.

- 4.** Modify the policies that use the filled up store to use the newly defined store.
- 5.** Unlock the original store.

New files checked into objects governed by these policies will be put in the new store. Files that were checked in previously can still be checked out and opened for view, but if new files are checked in, the new store is used. Refer to [*Implications of Changing Stores*](#) for more information, particularly if you are using a replicated environment.

Deleting a Store

If a file store is no longer required, you can delete it with the Delete Store statement:

```
delete store NAME;
```

NAME is the name of the file store to be deleted.

Deleting a file store deletes all file references within that store. Therefore, you should make a backup prior to deleting the store. This protects you if you need to access a file from that store at a later date. Once the store is deleted, you cannot restore it except through a backup.

When this statement is processed, eMatrix searches the list of existing file stores. If the name is found, that store is deleted along with all objects contained within that store. If the name is not found, an error message is displayed.

For example, to delete the Income Taxes store, enter the following MQL statement:

```
delete store "Income Taxes";
```

After this statement is processed, the store is deleted and you receive an MQL prompt for another statement.

Purging Files

The Purge Store statement is used to purge files from store locations. The syntax is:

```
purge store STORE_NAME [continue] [commit N] [location  
LOCATION_TARGET{,LOCATION_TARGET}];
```

If the Purge Store statement is used with just a store name, it does a purge on all locations, leaving files only in the default place defined in the store object. For example, if you wanted to purge all files in the store called Maps, you would enter the following MQL statement:

```
purge store Maps;
```

Optional clauses provide more control over which files are deleted. In addition, you can trace the purge event by using the `trace type store` command that logs the store's events. Refer to [Enabling trace tools via MQL](#) for more information.

Each `purge store` clause is described in the sections that follow.

continue clause

Include the keyword `continue` if you don't want the command to stop if an error occurs. If the log file is enabled, failures are listed in the file. Refer to [Enabling trace tools via MQL](#) for more information. For example:

```
purge store "Engineering-Dallas" continue;
```

If an error occurs when using the `continue` clause, the existing transaction is rolled back, so any database updates that it contained are not committed. The command starts again with the next business object. For this reason, when using the `continue` clause you should also include the `commit` clause, described below.

commit N clause

Include the `commit N` clause when purging large stores. The number N that follows specifies that the command should commit the database transaction after this many objects have been purged. The default is 10. For example:

```
purge store "Engineering-Dallas" continue commit 20;
```

location clause

To specify a subset of locations that you want purged, include the `location` clause. For example:

```
purge store "Engineering-Dallas" continue commit 20 location  
London,Paris,Milan;
```

When listing locations, delimit with a comma but no space.

6

Replicating Captured Stores

Captured Store Replication	148
Implications of Changing Stores.....	149
Defining a Location	150
Description Clause.....	150
Host Clause	150
Icon Clause.....	151
Protocol and Port Clauses.....	151
Password Clause.....	151
Path Clause	152
Permission Clause.....	152
User Clause	153
URL Clause	153
Hidden Clause	154
Property Clause	154
Modifying a Location Definition	155
Modifying a Location Definition.....	155
Deleting a Location.....	157
Purging Files.....	157
Defining Sites.....	158
Description Clause.....	158
Icon Clause.....	158
Member Clause	159
Hidden Clause	159
Property Clause	159
Modifying a Site Definition.....	160
Modifying a Site Definition	160
Deleting a Site	161
Synchronizing Captured Stores	162
Automatic Synchronization	162
Manual Synchronization	162
Using format.file	165
Tidying all locations	165

Captured Store Replication

Modeling of replicated captured stores relies on the creation of *locations* and *sites*.

- *Locations* provide alternate host and path information for captured stores. They can be added to existing captured store definitions only.
- A *site*, which is a set of locations, can be added to a person or group definition to specify location preferences.

Add the MX_SITE_PREFERENCE variable to the Collaboration Server's initialization file (bos.ini or ematrix.ini). This adjustment overrides the setting in the person or group definition for the site preference, and should be set to the site that is local to the server. This ensures optimum performance of file checkin and checkout for Web clients.

Captured stores can be distributed using *asynchronous replication*—data is duplicated to all mirrored locations only when commands are passed telling the database to do so. Therefore, when files are checked in, they are written to the location for the store that is also part of the user's preferred site, or if no match is found, the store's default location. They are not replicated to alternate locations until specifically told to do so.

The reason for introducing sites and locations is to enhance checkin, checkout, and open for view or edit performance for clients that require WAN access to a centralized storage location. The central store ("default") is mirrored to one or more remote machines ("locations"). In this way, a client at the remote site has LAN access to the data.

When a user performs a file checkin, it is written to that user's preferred location. If none is specified, eMatrix writes the file to the store's default path. In the eMatrix schema, the business object is now marked as having a file checked in at this location. When another eMatrix user requests the file for checkout, eMatrix will attempt the checkout from the following locations:

1. Locations that are part of the checkout person's preferred site. If none of these locations contains the newest copy, then;
2. The store's 'default' location. If this does not contain the newest copy, then;
3. Any location that contains a valid copy of the file. This means that a "sync on demand" is performed—the requested file is copied to the user's preferred location, and then the local file checkout is performed.

The files can be published to all locations associated with the store by running the MQL sync command against either the business object or the store. Following a synchronization, the files for a business object will be available in all locations. Refer to [Synchronizing Captured Stores](#) for more information.

Implications of Changing Stores

There may be times when you need to create a new store to replace an existing store. For example, the existing store may be running low on space. When you replace a store, you'll need to change the policies that reference the old store and have them reference the new store. That way, all new files that are checked into objects governed by the policies will be placed in the new store. Refer to [Monitoring Disk Space](#) for more information.

Be aware however that the old store will still be used because files checked in before the policy change will still reside in the old store. Also, if these objects are revised or cloned, the new revision/clone reference the original file and its storage location. When the time comes for the reference to become an actual file (as when the file list changes between the 2 objects) the file copy is made in the same store the original file is located in.

For systems that use replicated captured stores, files are copied for checkout and open operations to the locations and sites associated with a file's store.

Consider this example of a system that has replicated captured stores:

```
OldStore -> OldLocation1, OldLocation2  
NewStore -> NewLocation1, NewLocation2  
Site1: OldLocation1, NewLocation1  
Site2: OldLocation2, NewLocation2  
User1 default: Site1  
User2 default: Site2
```

1. User1 checks file into business object Rev1, which goes to OldLocation1.
2. Change policy to point to NewStore and lock OldStore.
3. User2 revises business object Rev1 to business object Rev2.
4. User2 checks out file in business object Rev2. File is created in OldLocation2 not NewLocation2.
5. User2 checks in file in business object Rev2. File gets created in NewLocation2. Note that copy of file in OldLocation2 made in Step 4 does not get deleted.

Defining a Location

Locations contain alternate host, path and FTP information for a captured store. The host, path and FTP information in the store definition is considered to be the *default* location for the store, while any associated location objects identify *alternate* file servers.

Think of a location as another store—it is defined as an FTP/NFS or UNC “location.” The same rules apply as in specifying a store.

An eMatrix location is defined with the Add Location Statement:

```
add location NAME [ADD_ITEM {ADD_ITEM}] ;
```

NAME is the name of the location you are defining. All locations must have a name unique in the database. The name can be up to 127 characters long and can contain spaces. You should assign a name that has meaning to both you and the user.

ADD_ITEM is an Add Location clause that provides more information about the location you are creating. The Add Location clauses are:

description VALUE
host HOST_NAME
icon FILENAME
protocol PROTOCOL_NAME
port PORT_NUMBER
password PASSWORD
path PATH_NAME
permission OWNER_ACCESS [,GROUP_ACCESS [,WORLD_ACCESS]]
user USER_NAME
url VALUE
[! not]hidden
property NAME [to ADMINTYPE NAME] [value STRING]

Description Clause

The description can provide general information for both you and the Business Administrator about the location. It can also help point out subtle differences between locations to the user.

Host Clause

The Host clause of the Add Location statement identifies the system that is to contain the location being defined. If a host is not specified, the current host is assumed and assigned.

If the location is to be physically located on a PC and accessed through a network drive, the host name must be set to `localhost`. For example:

```
add location host localhost
```

Icon Clause

The Icon clause of the Add Location statement associates an image with a file location. Icons help users locate and recognize items.

You can assign a special icon to the new location or use the default icon. The default icon is used when in view-by-icon mode. Any special icon you assign is used when in view-by-image mode. When assigning a unique icon, you must use a GIF image file. Refer to [Icon Clause](#) in Chapter 1 for a complete description of the Icon clause.

GIF filenames should not include the @ sign, as that is used internally by eMatrix.

Protocol and Port Clauses

When creating a location object for captured stores, you can include the parameters `protocol` and `port`.

Protocol and port can be defined when creating a location object for captured stores.

```
protocol PROTOCOL_NAME
```

`PROTOCOL_NAME` can be either `ftp`, `ftps`, or `nfs`.

You can enter `ftps` as a protocol value only if secure FTP is enabled on your system. See [Enabling Secure FTP for a Captured Store](#) in Chapter 5.

Each protocol has a default port that is used if not specified in the location definition. Include the `port` clause to specify a port other than the default.

```
port PORT_NUMBER
```

In support of this change, as of version 9, file names in error messages no longer use the eMatrix-specific format:

`/DIR/NAME@SERVER`

but will now use a Web-like URL:

`PROTOCOL:/STOREHOSTNAME:PORT/STOREPATH/FILENAME`

If a store/location does not have a protocol specified, eMatrix looks at other object attributes to determine which protocol was likely intended.

- If the host is ‘localhost’ (or empty), the protocol used will be ‘file’ (local file system).
- If the host is *not* ‘localhost’ and a username/password was given, then the store will use `ftp`.
- If the host is *not* ‘localhost’ and there is *no* username/password, the store will use `nfs`.

Note these checks eliminate the need to add protocol/port parameters to store/locations added prior to version 9.

Password Clause

The FTP password provides access to the FTP account.

The Password clause uses the following syntax:

```
password PASSWORD
```

where PASSWORD is the FTP password.

Before an FTP store location can be used, the location's host system must be configured to act as an FTP server, and the FTP Username and Password must be established. eMatrix has FTP client functionality built in so no additional software or configuration is necessary on the eMatrix workstations. See your operating system documentation for more information on installing and configuring FTP.

Path Clause

The path identifies where the location is to be placed on the host. When a location is defined, a directory for the captured files is created. If the captured store location will be accessed via FTP, the PATH_NAME can be relative to the FTP username login, but not necessarily the root directory of the FTP server system.

For example, you could use the following statement:

```
path Drawings
```

In this case, if an FTP login puts you on the host machine in the `usr/matrix` directory, the FTP store location would be `usr/matrix/Drawings`.

An absolute path can also be entered; however, the parent directory must already exist. For example, in the following statement, if the `/stores` directory does not exist on the target host, the location will not be created.:

```
path stores/store1
```

Permission Clause

Permissions specify read, write, or execute privileges for the new location. There are three categories of users:

- Owner
- Group
- World

These categories are assigned and controlled by your operating system. They are not the same as an Owner or Group within eMatrix; instead, Owner and Group are categories that define access (as described below).

The Permission clause uses the following syntax:

```
permission OWNER_ACCESS [,GROUP_ACCESS [,WORLD_ACCESS]]
```

For each category, you specify the type of permission:

r	Read access permission
w	Write access permission
x	Execute access permission

The default is `rw`, `rw`, `rw` which indicates Owner read/write access, Group read/write access, and World read/write access.

If you are specifying permission, you must include at least the Owner Access. If you choose to define Group Access or World Access, the Owner Access or Owner and Group Access must precede it, respectively.

For example, each of following definitions include a valid Permission clause:

```
add location "Manufacturing"  
    permission rw;  
  
add location "Personnel"  
    permission rw, r;  
  
add location "Library"  
    permission rw, rw, r;
```

In the first example, only the owner has read and write access to the location. In the second example, the owner has read and write access while the group has only read access. In the third example, both the owner and group may read and write, and everyone else has read access.

You must define at least the Owner Access. When setting the values for the permission, you need to know how ownership of the database files are assigned from within the operating system. If they were assigned to a single owner, you will want to set the Owner values for full access. If they were assigned to a group, the group should have full access.

The permission set is used only for NFS. FTP permissions are defined by users who logged in.

User Clause

When moving files to/from a store location, the FTP username defines the FTP account used.

The User clause uses the following syntax:

```
user USER_NAME
```

where USER_NAME is the FTP username.

Before an FTP store location can be used, the location's host system must be configured to act as an FTP server, and the FTP Username and Password must be established. eMatrix has FTP client functionality built in so no additional software or configuration is necessary on the eMatrix workstations. See your operating system documentation for more information on installing and configuring FTP.

URL Clause

To implement URL text searching, a URL is stored with eMatrix Location objects. The URL is in the form of a CGI-bin request. It is assumed that the CGI-bin program and the indexing engine used by the URL is installed and administered independently of eMatrix.

Each Location should specify the following URL:

`http://${HOST}/matrix/mxis.asp?ct=${LOCATION}&qu=${QUERY}&mh=${LIMIT}`

During query evaluation, these variables are expanded as follows:

- LOCATION is expanded to be the name of the eMatrix location object
- HOST is expanded to contain the host of the eMatrix location object
- QUERY is expanded to contain the search string
- LIMIT is expanded to contain the contents of the MX_FULLTEXT_LIMIT variable in each users' initialization file

If you are using FTP locations, the paths will vary from the paths used to create the directories for the index server catalog.

Initialization File Settings

Some search engines may need more information than provided with the basic URL shown above. Additional common parameters to be used with the full text search integration can be set in the initialization file and added to the URL specified in the location. For example, MODE, LIMIT, and MINSCORE settings can be passed to a search engine by setting the following variables in the initialization file:

- MX_FULL_TEXT_MODE
- MX_FULL_TEXT_LIMIT
- MX_FULL_TEXT_MINSCORE

The URL should be modified using variable syntax. For example, to add the MINSCORE setting use:

```
http://${HOST}/matrix/mxis.asp?ct=${LOCATION}&qu=${QUERY}&mh=${LIMIT}&score=${MINSCORE}
```

The possible values for these variables depends on the search engine used. However, many have the concepts of a simple versus a complex *MODE*, a *LIMIT* to the number of files to return, and a *MINSCORE* value which the file must meet. Also, up to three additional parameters can be passed to the search engine using the ARG1, ARG2, and ARG3 variables in the initialization file.

If you set any of these variables in the initialization file and add them to the URL of the store, eMatrix will provide them to the search engine, but the search engine will then be responsible for interpreting the settings.

Refer to *Enabling Text Searches on Captured Stores* in the *eMatrix System Manager Guide* for more information.

Hidden Clause

You can specify that the new location object is “hidden” so that it does not appear in the Location chooser in eMatrix. Users who are aware of the hidden location’s existence can enter its name manually where appropriate. Hidden objects are accessible through MQL.

Property Clause

Integrators can assign ad hoc attributes, called Properties, to the location. Properties allow associations to exist between administrative definitions that aren’t already associated. The property information can include a name, an arbitrary string value, and a reference to another administration object. The property name is always required. The value string and object reference are both optional. The property name can be reused for different object references, that is, the name joined with the object reference must be unique for any object that has properties.

```
add location NAME  
property NAME [to ADMINTYPE NAME] [value STRING];
```

For additional information on properties, see *Overview of Administration Properties* in Chapter 24.

Modifying a Location Definition

Modifying a Location Definition

After a location is defined, you can change the definition with the Modify Location statement. This statement lets you add or remove defining clauses and change the value of clause arguments:

```
modify location NAME [MOD_ITEM {MOD_ITEM}];
```

NAME is the name of the location to modify.

MOD_ITEM is the type of modification to make. Each is specified in a Modify Location clause, as listed in the following table. Note that you need to specify only fields to be modified.

Modify Location Clause	Specifies that...
description VALUE	The description is changed to the new value specified.
host HOST_NAME	The host name is changed to the value specified.
icon FILENAME	The image is changed to the new image in the file specified.
protocol PROTOCOL_NAME	The protocol is changed to the value specified.
port PORT_NUMBER	The port is changed to the number specified.
name NAME	The location name is changed to the new name.
password PASSWORD	The FTP password is changed to the value entered.
path PATH_NAME	The current comment, if any, is changed to the value entered.
permission OWNER_ACCESS [,GROUP_ACCESS [,WORLD_ACCESS]]	Permissions for the location are changed to the values specified.
user USER	The FTP username is changed to the name specified.
url VALUE	The current URL is changed to the new one entered.
hidden	The hidden option is changed to specify that the object is hidden.
nothidden	The hidden option is changed to specify that the object is not hidden.
property NAME [to ADMINTYPE NAME] [value STRING]	The named property is modified.
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.

As you can see, each modification clause is related to the clauses and arguments that define the location.

When you modify a location, you first name the location and then list the modifications. For example, the following statement changes the permissions of the NY-Documents location:

```
modify location "NY-Documents"  
    permission rw, r, r;
```

When this statement is executed, the permissions of the location become read/write for the owner and read-only for group and world.

Deleting a Location

If a location becomes obsolete, you can delete that location by using the Delete Location statement:

```
delete location NAME;
```

NAME is the name of the location to be deleted.

When this statement is processed, eMatrix searches the list of defined locations. If the name is not found, an error message is displayed. If the name is found, the location is deleted.

Purging Files

The Purge Location statement removes all files from the location. For example, if you wanted to purge all files in the location called Shelton, you would enter the following MQL statement:

```
purge location Shelton;
```

Purge removes the metadata for all the location files and then deletes the location files.

If a location holds the only valid copy of a file, then that file is replicated back to the store's default location (the place defined in the store object itself).

Removing a location from a store will perform a purge location before disassociating the location from the store. Therefore the files are replicated to the store before the location is removed from the store and the business objects will not lose their files.

Defining Sites

Sites are nothing more than a set of locations. A site can be associated with a person or group object. When associated with a person, the site defines the list of locations preferred by a particular person. When associated with a group, the site defines the list of locations preferred by all members of the group.

In addition, a matrix.ini, bos.ini, or ematrix.ini file may contain the following setting:

```
MX_SITE_PREFERENCE = SITENAME
```

where SITENAME is the name of the site object.

This setting overrides the setting in the person or group definition for the site preference. It is particularly designed for use in the bos.ini, where all Web clients should use the site preference of the eMatrix Collaboration Server to ensure optimum performance. Refer to the *eMatrix Installation Guide* for more information.

An eMatrix site is defined with the Add Site Statement:

```
add site NAME [ADD_ITEM {ADD_ITEM}]
```

NAME is the name of the site you are defining. All sites must have a name unique in the database. The name can be up to 127 characters long and can contain spaces. You should assign a name that has meaning to both you and the user.

ADD_ITEM is an Add Site clause that provides more information about the site you are creating. The Add Site clauses are:

description VALUE
icon FILENAME
member location NAME
[! not]hidden
property NAME [to ADMINTYPE NAME] [value STRING]

Description Clause

The description can provide general information for both you and the Business Administrator about the site. It can also help point out subtle differences between sites to the user.

Icon Clause

The Icon clause of the Add Site statement associates an image with a file store. Icons help users locate and recognize items.

You can assign a special icon to the new site or use the default icon. The default icon is used when in view-by-icon mode. Any special icon you assign is used when in view-by-image mode. When assigning a unique icon, you must use a GIF image file. Refer to [Icon Clause](#) in Chapter 1 for a complete description of the Icon clause.

GIF filenames should not include the @ sign, as that is used internally by eMatrix.

Member Clause

Use the Member Clause of the Add Site statement to add locations to the site definition. The location named must exist in the database and the spelling and case must match exactly. If the location name contains embedded spaces, use quotation marks. For example:

```
add site member location "Western Division";
```

Hidden Clause

You can specify that the new site is “hidden” so that it does not appear in the Site chooser in eMatrix. Users who are aware of the hidden site’s existence can enter its name manually where appropriate. Hidden objects are accessible through MQL.

Property Clause

Integrators can assign ad hoc attributes, called Properties, to the site. Properties allow associations to exist between administrative definitions that aren’t already associated. The property information can include a name, an arbitrary string value, and a reference to another administration object. The property name is always required. The value string and object reference are both optional. The property name can be reused for different object references, that is, the name joined with the object reference must be unique for any object that has properties.

```
add site NAME  
    property NAME [to ADMINTYPE NAME] [value STRING];
```

Modifying a Site Definition

Modifying a Site Definition

After a site is defined, you can change the definition with the Modify Site statement. This statement lets you add or remove defining clauses and change the value of clause arguments:

```
modify site NAME [MOD_ITEM {MOD_ITEM}];
```

NAME is the name of the site to modify.

MOD_ITEM is the type of modification to make. Each is specified in a Modify Site clause, as listed in the following table. Note that you need to specify only fields to be modified.

Modify Site Clause	Specifies that...
name NAME	The site name is changed to the new name.
description VALUE	The description is changed to the new value specified.
icon FILENAME	The image is changed to the new image in the file specified.
add location NAME	The named location is added to the site definition.
remove location NAME	The named location is removed from the site definition.
hidden	The hidden option is changed to specify that the object is hidden.
nothidden	The hidden option is changed to specify that the object is not hidden.
property NAME [to ADMINTYPE NAME] [value STRING]	The existing property is modified according to the values specified.
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.

As you can see, each modification clause is related to the clauses and arguments that define the site.

Deleting a Site

If a site is no longer required, you can delete it. Since deleting a site affects all locations within that site, it is recommended that you make a backup prior to deletion. This protects you if you need access to files from that site at a later date. You may need to involve the DBA in the backup and restore process (if a restore is necessary).

To delete a site, use the Delete Site statement:

```
delete site NAME;
```

NAME is the name of the site to be deleted.

When this statement is processed, eMatrix searches the list of defined sites. If the name is not found, an error message is displayed. If the name is found, the site is deleted.

Synchronizing Captured Stores

Synchronization is the means that eMatrix uses to ensure that users access the latest versions of files when checking out or viewing files in a replicated store environment. Synchronization occurs in one of two ways:

- eMatrix automatically synchronizes files for a user's preferred location during checkout and open for viewing operations.
- System Administrators can use MQL commands to synchronize files for a business object or for a store to ensure that all locations contain the most recent copy of the files.

If you use FTP via the command line to transfer hashed file names and then run a sync command, eMatrix will ignore the files that you have copied or put there. eMatrix names the files as it creates the copies and stores the names in the database, all as part of the sync operation.

Automatic Synchronization

Synchronization occurs automatically when a person checks out or opens a file for viewing and the file at the user's preferred location is not the latest version. In such a case, eMatrix copies the latest version of the file to the user's preferred location and then checks out or opens the local file.

Manual Synchronization

In order to publish newly checked-in files to other locations associated with the store, System Administrators must use the `sync` statement against either the business object or the store. In such a case, eMatrix performs a *sync on demand*, using one of the following:

- To synchronize using the business object, use the following command:

```
sync businessobject TYPE NAME REV [to [store] [location  
LOCATION_TARGET{,LOCATION_TARGET}]] [from [store][location  
LOCATION_SOURCE{,LOCATION_SOURCE}]] [update] [overwrite];
```

This command copies the business object's files to the specified locations.

- To synchronize using the store name, use the following command:

```
sync store STORE_NAME [continue] [commit N] [to [store]  
[location LOCATION_TARGET{,LOCATION_TARGET}]] [from  
[store][location LOCATION_SOURCE{,LOCATION_SOURCE}]] [update]  
[overwrite];
```

All files within the named store are copied to the specified locations.

When comparing files between a remote and host store, eMatrix looks for the hashed name that is known to the database, for the business object associated with the store or location, as well as the file size and permissions.

The `sync` command clauses and related `sync` commands are described in the sections that follow.

continue clause (sync store only)

Include the keyword `continue` if you don't want the command to stop if an error occurs. If the log file is enabled, failures are listed in the file. Refer to [Enabling trace tools via MQL](#) for more information.

For example:

```
sync store "Engineering-Dallas" continue
```

If an error occurs when using the `continue` clause, the existing transaction is rolled back, so any database updates that it contained are not committed. The command starts again with the next business object. For this reason, when using the `continue` clause you should also include the `commit` clause, described below.

commit N clause (sync store only)

Include the `commit N` clause when synching large stores. The number "N" that follows specifies that the command should commit the database transaction after this many objects have been synched. The default is 10. For example:

```
sync store "Engineering-Dallas" continue commit 20
```

to clause

If you want to sync only a subset of all of a store's locations, include the `to location` and/or `store` clause. For example:

```
sync bus Assembly R123 A to location London,Paris,Milan store;
```

```
sync store "Engineering-Dallas" continue commit 20 to location  
London,Paris,Milan store;
```

When specifying only some locations, a comma (but not a space) is used as a separator. Including the keyword `store` indicates that the store itself (default host and path) should be updated, too.

If an on-demand sync is initiated for multiple locations, the sync will be rolled back if the sync fails at any of the locations. For example, assume an on-demand sync for five locations. The first two locations sync, but the third fails because the machine is down. The fourth and fifth stores will probably not get synched, depending on where the file is physically located and what order the stores are created in.

- If the file lives in a downed store, none will get synced.
- If the file lives in a store that is online, then the sync will fail when it reaches the downed store. Any stores that would have been synced after that store will not be synced.

While the stores that were synced will retain the new file, eMatrix will rollback the sync command and think that only the original location of the file will still have that file. eMatrix will not know of any other locations that now have the file.

from clause

Use the **from** clause to specify locations from which files will be replicated to other specified locations. For example:

```
sync store "Engineering-Dallas" continue commit 20 from
location London,Paris,Milan to location Moscow,Boston;
```

```
sync bus Assembly R123 A from location London,Paris,Milan
store;
```

If you do not include the **to** keyword, all locations will be updated, potentially including those listed in the **from** clause. You can optionally include the **store** keyword in either the **from** or **to** clause to include the store's host and path.

If an on-demand sync is initiated for multiple locations, the sync will be rolled back if the sync fails at any of the locations, as described above in [to clause](#).

update clause

Use the **update** clause to copy files only to locations that contain a previous version of the file. For example:

```
sync store "Engineering-Dallas" continue commit 20 update
```

```
sync bus Assembly R123 A update;
```

overwrite clause

The **overwrite** clause, used with the **to|from** location and/or **store** clauses, forces an overwrite of files on those servers. All files are copied to the specified locations/store without doing a comparison. For example:

```
sync store "Engineering-Dallas" continue commit 20 from
location London,Paris,Milan to location Moscow,Boston
overwrite;
```

```
sync bus Assembly R123 A from location London,Paris,Milan store
overwrite;
```

sync businessobjectlist

Use the **sync businessobjectlist** clause to specify a list of business objects to sync:

```
sync businessobjectlist BUS_OBJECT_COLLECTION_SPEC [continue
[commit N] [to [store] [location
LOCATION_TARGET{,LOCATION_TARGET}]] [from [store] [location
LOCATION_SOURCE{,LOCATION_SOURCE}]] [update] [overwrite];
```

BUS_OBJECT_COLLECTION_SPEC includes:

```
| set NAME
| query NAME
| temp set BO_NAME{,BO_NAME}
```

```
| temp query [TEMP_QUERY_ITEM {TEMP_QUERY_ITEM}]  
| expand [EXPAND_ITEM {EXPAND_ITEM}]
```

You can also use these specifications in boolean combinations. Refer to the [Expressions on Queries](#) in Chapter 38 for more information on evaluating expressions.

The keyword `businessobjectlist` can be shortened to `buslist`.

For example:

```
sync buslist set MyAssemblies continue commit 20 to location  
London,Paris,Milan store;
```

Using `format.file`

The proper handling of distributed file stores requires that you be able to identify which of several locations holds up-to-date versions of a particular file checked into a given business object.

The `file.format` select clauses shown below are available to more efficiently manage on demand synchronization. Using the `print bus TNR select` command, you can find all locations that hold up-to-date versions of a particular file in order to sync files using a specific location.

For example, suppose you have three locations over a WAN and a sync needs to happen to one location because a user is requesting the file there. You can force sync from the “nearest” synced location, that is, the location from which the FTP traffic is fastest.

`format.file.location`

Finds locations which are in the current user’s preferred sites *and* are synchronized, and returns the first of them. If no locations satisfy both conditions, returns the first synchronized one.

`format.file.synchronized`

Returns all locations holding synchronized versions of the checked in file(s), regardless of user preference.

`format.file.obsolete`

Returns all locations holding a copy of the file that has been marked as obsolete (needing synchronization), regardless of user preference.

When using `format.file[FILENAME].*`, you can use either of the following formats for the filename:

- the complete host:/directory/filename.ext from which it was last checked in.
- just the filename (`FILENAME.EXT`), which must be unique within this object/format. This makes the extraction of data specific to a single file much easier.

Tidying all locations

In replicated environments, when a user deletes a file checked into an object (via checkin overwrite or file delete), by default all other locations within that store maintain their copies of the now obsolete file. You can run the `tidy store` command to remove obsolete copies.

To remove obsolete files at each location

- Use the following statement:

```
tidy store NAME;
```

You can change the system behavior going forward such that all future file deletions occur at all locations by using the `set system tidy on` command. Since this command changes future behavior and does not cleanup existing stores, you should then sync all stores, so all files are updated, and then tidy so obsolete files are removed at all locations. Once done, the system will remain updated and tidy, until and unless system tidy is turned off.

Running with system tidy turned on may impact the performance of the delete file operation, depending on the number of locations and network performance at the time of the file deletion.

Distributing the Database

Distributed and Replicated Data	168
Preparing for Distribution	169
System Setup and Installation	169
Working with Servers	171
Defining a Server	171
Description Clause.....	172
Icon Clause.....	172
User Clause.....	172
Password Clause.....	172
Connect Clause	172
Timezone Clause.....	172
Hidden Clause	175
Property Clause	175
Modifying a Server Definition	176
Modifying a Server Definition.....	176
Disabling or Deleting a Server.....	177
Disabling a Server	177
Deleting a Server	177
Distributing Data Across Servers	178
Setting Up a Distributed Database	178
Considerations.....	180
Network Stability.....	180
Locating the Master Vault.....	180
Renaming Servers	180
Distribution Problems.....	180
Importing Servers	181
Should I Use a Link or a Copy?	181
Using Schema Names	183
Requirements	183
Use Case	183

Distributed and Replicated Data

Modeling of distributed data relies on the creation of several kinds of administration objects. eMatrix business object data is contained in *Vaults*. Any associated files for these objects are kept in *Stores*. Vaults and Ingested Stores will have corresponding table columns in the relational database engine, which are located on Oracle *Servers*. A database may consist of data located on many servers in various *Locations* within a *Site*. The database may even be disbursed across several sites. A single database with one set of administrative definitions, no matter how distributed, is called a *Federation*.

Ideally, users most often require access to the information contained on the server located closest to them. However, through the use of various distribution and replication techniques, access to all information in a Federation is possible. You can:

- Replicate captured stores, as described in *Captured Store Replication* in Chapter 6.
- Distribute a database across multiple servers, as described in *Distributing Data Across Servers*.
- Use the Server Distribution Table to replicate or “link distribute” vaults and ingested stores. This type of distribution requires a special, additional license from MatrixOne.
- Create “Remote” vaults from a separate federation to add to your database. This allows partners to share data, even if the schema relating to the common objects is slightly different. This is described in *Sharing Data Between Federations* in the *eMatrix System Manager Guide*.
- Create and/or use AdapletsTM and “Foreign” vaults to add data from a completely different database system. Consult your eMatrix Professional Services representative or Customer Service for more information.

Preparing for Distribution

While setting up a distributed database is easily executed from eMatrix System Manager, it requires careful preparation by the System and Database Administrator(s). When distributing an eMatrix database across a WAN, or even within one location across several Oracle servers, communication between all servers must first be established at the database level. The following guidelines should be used in preparation for distributing an eMatrix database.

Currently you cannot distribute eMatrix/DB2 databases.

System Setup and Installation

Obviously, the servers must have the Oracle recommended O/S version installed. While setting up replication and distribution, it may be helpful to change the system default TCP/IP parameter for clearing a port. For example, on Solaris, the default is 4 minutes. To see this, use the command:

```
%ndd /dev/tcp tcp_close_wait_interval
```

It should return a value of 240000 by default. To change it to 20 seconds, use the command:

```
%ndd -set /dev/tcp tcp_close_wait_interval 20000
```

This should greatly affect the ability to start a process in a timely manner after stopping it, which may need to be done frequently when setting up and testing distribution.

Oracle

The Oracle Enterprise Server Software must be installed on each server. A database instance should be created by the DBA on each server. Consult the *eMatrix Installation Guide* and *Defining Data and Index Table Space* in the *eMatrix System Manager Guide* for guidelines for initialization parameter settings, tablespace sizes, and location of data files, log files and control files. In addition, add or modify the following setting in the init<SID>.ora file in all instances:

```
global_names=false
```

If you have more than five servers in your distribution schema, you will need to modify the following statement in the init<SID>.ora file at all participating servers:

```
open_links = 50
```

The default is 4; this restricts the number of db_links to 4. The number of db_links is always one less than the number of servers configured.

Oracle Users and Tablespaces

For each Oracle instance or server, you must create an Oracle User and assign a name and password that have meaning to your application. Oracle User names need not be identical across a distributed Federation.

All tablespace names on each Oracle instance where a vault will be copied must have the same names as the tablespace names used for the Master vault.

eMatrix servers must adhere to Oracle naming conventions. This means that Server names (as well as Oracle user names) cannot begin with a number. If they do, you will receive the following error when starting eMatrix:

```
ORA-00987: missing or invalid usernames
```

However, eMatrix Server and Oracle user names may *contain* a number (for example, mx7010).

Connect_Strings

Create Oracle database aliases (*connect string* entries for eMatrix) at all servers for all other servers. On Windows this is done using SQL*Net Easy Configuration. Carefully note the spelling and case of each entry so that eMatrix connect strings in the server definition will be correct.

It is essential that each server in a distribution schema is uniquely identified by the exact same connect string at all participating servers. Server names must NOT contain embedded spaces.

Verification

Verify using SQL*Plus that each server can connect to each other server, using the user account set up for eMatrix. This confirms the connect string and the Oracle User.

Once the Oracle portion is configured correctly, an eMatrix client machine should be set up and connected to one of the Oracle servers. At this point, the modeling of the servers in eMatrix can begin.

Network

When using synchronized database replication, it is strongly recommended that the network be stable. We suggest using redundant network paths between synchronized replicated servers. It is less critical that the network be fast; stability is essential.

Working with Servers

In order to distribute a database across a WAN, several Oracle instances are created. Each Oracle instance will correspond to an eMatrix server object. The eMatrix server parameters are:

- Username
- Password
- Connect String
- Time Zone

Based on these settings, users' connection files can point to any of the servers in the federation, ideally the one that is physically located the closest. The first three parameters (Username, Password, and Connect String) establish the Oracle instance that contains the schema. The Time Zone setting is important when eMatrix databases are used throughout the world. It enables users to see dates and times based on a conversion to their local time zone. Refer to [Time Zone Usage](#) for more information.

It is *highly recommended* that a Server is created for all eMatrix databases, even those that use only Local vaults. It is *required* when Remote or Foreign vaults are added, or when distributing vaults as described in [Distributing Data Across Servers](#).

Currently you cannot distribute eMatrix/DB2 databases. However, server objects may be created to establish a Time Zone.

For administrative work done in the Business, System, or MQL applications, all Server objects are accessed. For this reason, machines used for these functions must have an Oracle database alias configured for each server object. For example, for a system with two Servers with connectstring values of "SanDiego" and "SanJose", administrator's machines must have two Oracle database aliases configured named "SanDiego" and "SanJose". Refer to Chapter 2, *Installing the Database* in the *eMatrix Installation Guide* for information on adding database aliases.

Defining a Server

An eMatrix server is defined with the Add Server statement:

```
add server NAME [ ITEM {ITEM} ];
```

NAME is the name of the server you are defining. All servers must have a unique name. The server name is used as the name of the Oracle link that is created, so it must conform to Oracle naming conventions. **Spaces are not allowed. Server names cannot begin with a number.** They can, however, contain numbers. Consult Oracle documentation for naming conventions.

ITEM is an Add Server clause that provides more information about the server you are creating. The Add Server clauses are:

description VALUE
icon FILENAME
user USER_NAME
password PASSWORD

```
connect CONNECT_STRING  
timezone ZONE  
[ ! |not ]hidden  
property NAME [ to ADMINTYPE NAME ] [ value STRING ]
```

Description Clause

The description can provide general information about the purpose of the server. It can also help point out subtle differences between servers to the user.

Icon Clause

Icons help users locate and recognize items. You can assign a special icon to the new server or use the default icon. The default icon is used when in view-by-icon mode. Any special icon you assign is used when in view-by-image mode. When assigning a unique icon, you must use a GIF image file. Refer to [Icon Clause](#) in Chapter 1 for a complete description of the Icon clause.

GIF filenames should not include the @ sign, as that is used internally by eMatrix.

User Clause

Use the User clause of the Add Server statement to specify the Oracle User that has been created for use with eMatrix. It was created by the Oracle DBA, as described in [Preparing for Distribution](#). The User clause uses the following syntax:

```
user USER_NAME
```

Password Clause

The Password clause of the Add Server statement is used to specify the password that is associated with the Oracle User. It was defined by the Oracle DBA at the time of User creation, as described in [Preparing for Distribution](#).

Connect Clause

The Connect clause of the Add Server statement is used to specify the Oracle database Alias. The Oracle Alias was created on the Oracle server by the Oracle DBA. The Oracle Alias and the connect string entry must exactly match in spelling and case. This is especially important in a mixed UNIX and NT environment. The Connect clause uses the following syntax:

```
connect CONNECT_STRING
```

Timezone Clause

The Timezone clause of the Add Server statement is used to specify the default timezone for the server. The Timezone clause uses the following syntax:

```
timezone ZONE
```

ZONE is a time zone supported by eMatrix, as shown in the table below. You can use either the “in relation to GMT” value or the abbreviation. For example, for Eastern Standard Time, type either **EST** or **GMT-5**.

Time Zones Supported by eMatrix		
In relation to GMT	Abbreviation	Meaning
GMT-12		
GMT-11		
GMT-10	HST, HDT	Hawaiian Standard or Daylight time
GMT-9	YST, YDT	Yukon Standard or Daylight time
GMT-8	PST, PDT	Pacific Standard or Daylight time
GMT-7	MST, MDT	Mountain Standard or Daylight time
GMT-6	CST, CDT	Central Standard or Daylight time
GMT-5	EST, EDT	Eastern Standard or Daylight time
GMT-4	AST, ADT	Atlantic Standard or Daylight time
GMT-3		
GMT-2		
GMT-1		
GMT		
GMT+1	MET, MET DST	Mediterranean Standard or Daylight time.
GMT+2	EET, EET DST	Eastern european Standard or Daylight time
GMT+3		
GMT+4		
GMT+5		
GMT+6		
GMT+7		
GMT+8	WST	
GMT+9	JST	Japanese Standard time
GMT+10		
GMT+11		
GMT+12	NZST	New Zealand Standard time

Note that time zones can be set in eMatrix Server objects with either the “in relation to GMT” value, or by their abbreviations. However, use of the abbreviation is recommended since it will take into account the Daylight Savings time switch, while the offset value would have to be manually changed when moving between Daylight and Standard times.

Time Zone Usage

eMatrix handles time zones by converting all times to Greenwich Mean Time (GMT) at the time of creation, and converting them back to a user's local time whenever they are being displayed. This accomplishes two goals:

- Converting all times into GMT in the database makes time comparisons legitimate.
- Converting all times into the user's local time for display is easier for the user.

Times that are generated by the system are handled differently than user keyed-in times.

- *System generated times* (originated, modified, history, actual, mail) are generated with the database server's clock, but they are converted to GMT for storing, and can therefore be converted properly to the user's local time zone for display.
- *User keyed-in times* (attribute, scheduled) are converted to GMT on input and thereafter converted properly to the user's local time zone for display.

This will allow the eMatrix client, wherever it is, to display all times in its local time zone. For example, assume a company has clients in San Jose, Boston, and Bonn, all set to their local time zones. A user in San Jose promotes an object on December 11, 1998, 4:45 pm (PST). Clients in Boston will read the Actual Date in EST (December 11, 1998, 7:45 pm) and clients in Bonn will read it in German local time (December 12, 1998, 1:45 am).

Establishing a Client Time Zone

In order for history and other times to be converted to GMT for storage, all client machines must have a time zone established. Windows platforms must have a time zone specified by default. However, it is possible that it is set inappropriately and so it is recommended that all time zones are validated. Consult Windows Help from the Start menu to make any necessary adjustments.

On UNIX platforms, the user is prompted to enter a time zone during eMatrix installation. If the system's environment has a time zone established, it will be displayed as the default value of the time zone variable, and the user can simply “Enter” though the prompt. However, if the time zone environment variable is not set on the UNIX machine, the prompt will default to GMT, and the user should enter the correct time zone. This environment setting will be added to the eMatrix start-up script, and used by eMatrix.

When times are recorded in history for a business object, they are offset from the client's time to server time by an offset amount calculated at the time eMatrix starts. Even if the client's clock changes, the offset is still applied and is not adjusted. (However, an adjustment is made by eMatrix for changes between standard and daylight time.) This offset can result in wrong times being recorded in history. For example, if after starting eMatrix, the client's clock is put back by one hour, new times recorded in history will be one hour earlier than the current time on the server. Because of this, *it is recommended that if changes are required to a system clock, eMatrix should be shut down before the time adjustments are made.*

Hidden Clause

You can specify that the new server is “hidden” so that it does not appear in the Server chooser in eMatrix. Users who are aware of the hidden server’s existence can enter its name manually where appropriate. Hidden objects are accessible through MQL.

Property Clause

Integrators can assign ad hoc attributes, called Properties, to the server. Properties allow associations to exist between administrative definitions that aren’t already associated. The property information can include a name, an arbitrary string value, and a reference to another administration object. The property name is always required. The value string and object reference are both optional. The property name can be reused for different object references, that is, the name joined with the object reference must be unique for any object that has properties.

```
add server NAME  
    property NAME [to ADMINTYPE NAME] [value STRING];
```

For additional information on properties, see *Overview of Administration Properties* in Chapter 24.

Modifying a Server Definition

Modifying a Server Definition

After a server is defined, you can change the definition with the Modify Server statement. This statement lets you add or remove defining clauses and change the value of clause arguments:

```
modify server NAME [MOD_ITEM {MOD_ITEM}];
```

NAME is the name of the server to modify.

MOD_ITEM is the type of modification to make. Each is specified in a Modify Server clause, as listed in the following table. Note that you only need to specify fields to be modified.

Modify Server Clause	Specifies that...
description VALUE	The description is changed to the new value specified.
icon FILENAME	The image is changed to the new image in the file specified.
user USER_NAME	The Oracle username which has been created for use with eMatrix is changed to the name specified.
password PASSWORD	The password that is associated with the Oracle User is changed to the value entered.
connect CONNECT_STRING	The eMatrix Connect String (which matches the Oracle database Alias) is changed to the value entered.
timezone ZONE	The named timezone is modified.
master vault VAULT_NAME	The name of the master vault is changed to the new name specified. See Distributing Data Across Servers .
copy vault VAULT_NAME	The named vault is copied to the server from the master server. See Distributing Data Across Servers .
link vault VAULT_NAME	The named vault is linked to the server. See Distributing Data Across Servers .
hidden	The hidden option is changed to specify that the object is hidden.
nothidden	The hidden option is changed to specify that the object is not hidden.
property NAME [to ADMINTYPE NAME] [value STRING]	The named property is modified.
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.

As you can see, each modification clause is related to the clauses and arguments that define the server.

Disabling or Deleting a Server

Disabling a Server

Servers in a distributed environment can be disabled to prohibit access to them. Use the following MQL command:

```
disable server NAME;
```

Users will not be able to log on to a disabled server, nor will vaults that are mastered there be available. This is helpful following the failure of a machine or network, to allow access in a distributed database. Once the failed systems have been disabled, the surviving servers can still be used.

After being disabled, the server is taken out of distribution. To re-enable it, use the Server Distribution Table.

Deleting a Server

If a server is no longer required, you can delete it with the Delete Server statement:

```
delete server NAME;
```

NAME is the name of the server to be deleted.

Deletion of a server does not delete vaults.

When this statement is processed, eMatrix searches the list of existing servers. If the name is found, the server is deleted. If the name is not found, an error message is displayed.

Distributing Data Across Servers

When multiple eMatrix servers are created, vaults and ingested stores can be distributed or replicated to another server in the federation. This is done by editing the Server Distribution Table.

Currently you cannot distribute eMatrix/DB2 databases.

A database can be distributed across multiple Oracle servers in two ways:

- One method is to have a vault mastered at one server with links at the servers pointing back to the master. The Oracle tables physically reside on this “master” server. Other servers access these tables using SQL*Net database links. This is called *distribution*. This makes sense for the case where metadata is divided along vault lines where a majority of people at only one location access this data.
- In the second method, *synchronous replication*, a vault is mastered at one server but copied at another. The database tables physically reside at all servers. Any updates to either server is instantly transmitted to all servers on a 2 phase commit basis. This permits local read/write access to all metadata in that vault to all servers involved. Update performance will be a little slower as all servers need to be updated.

Within eMatrix, vaults and ingested stores are essentially the same—an eMatrix storage location within an Oracle table. In the discussions that follow, when the term vault is used, take it to mean both vaults and ingested stores.

Setting Up a Distributed Database

Vaults should be distributed while logged on to the server that contains the master, or original, data. Data can be distributed across servers with the `modify server` statement.

Master Clause

The master server for a vault contains the original Oracle tablespaces for it. Only one server can be listed as the master for a vault. The following example assigns the Morocco vault as the master vault for the server Jupiter:

```
modify server Jupiter
    master vault Morocco
```

Link Clause

A server can connect to a vault located on another server via a link. This is called *link distribution*. The Oracle Tables on the master server are accessible to the linked server. Any number of servers can be linked to a vault on the master server. The following example allows the server Pluto to connect to the vault Copenhagen.

```
modify server Pluto
    link vault Copenhagen
```

Copy Clause

A vault can be copied onto other servers from the master server. This is called *synchronous replication* — the Oracle Tables are replicated on the “copy” server. Any number of servers can contain a copy of a vault from the master server. The following example copies the vault Oslo from the master server to the server named Neptune.:

```
modify server Neptune
    copy vault Oslo
```

Vaults should be distributed and replicated while logged on to the master vault's Oracle instance.

Considerations

As stated before, when distributing databases, careful planning is required. The following sections describe some issues which should be considered when planning the database distribution schema.

Network Stability

Ensure that the network is stable when considering using synchronous replication—low network bandwidth is less of a negative factor than network instability. This network distributed database architecture requires redundant network routes between servers. The database can be disabled for all users if either server goes down, or if the servers can not communicate with each other. Uninterruptable power supplies should be employed to protect the network.

If a server or network failure occurs for a synchronized replicated vault, the vault will still be locally accessible but only in read-only mode until server/network restoration.

Locating the Master Vault

The Oracle server which is located in the LAN with the most eMatrix clients should be set up as the Master database server. Since the federation is only as reliable as its weakest server, it should not be presumed that the master server should be the strongest or best maintained server, if this is in a central office remote from the users. All servers must be reliable and well maintained, and proximity to the users should be the overriding factor in deciding where to locate the master server.

When modifying the Distribution Table to set up links or copies of a vault, the connection (bootstrap) file of the client machine performing the modification should be pointing at the server where the Master is located.

You can set a vault to be mastered at any server in the distribution federation. A vault may be mastered at one server; another vault may be mastered elsewhere.

Renaming Servers

If it is necessary to rename a machine that is part of a distribution schema, modify the database alias rather than define a new one. Simply modify the connect string Alias to point to the newly named server.

Distribution Problems

Distribution problems are typically a result of network or SQL*Net configuration issues. Prior to distribution, confirm that all servers can assess each other's Oracle user/password/connect strings via SQL*Plus. Also edit all server definitions; this confirms that the server is accessible from the current server.

If a server in a distributed database is lost, or an Oracle user deleted, changes cannot be made to the database until you disable the failed server. To prohibit access to the failed server, use the following MQL command:

```
disable server NAME;
```

Users will not be able to log on to a disabled server, nor will vaults that are mastered there be available. Once the failed systems have been disabled, the surviving servers can still be used.

After being disabled, the server is taken out of distribution. To re-enable it, use the Server Distribution Table.

Importing Servers

If a server object is imported (via MQL or Oracle) into a different Oracle instance or user than it was exported from, the username and password settings of the server must be modified before distributed access is available. This is particularly important if an Upgrade will follow the import, since all servers must be accessible to the machine doing the upgrade.

Should I Use a Link or a Copy?

Both link distribution and synchronous replication have their merits and downfalls when creating a global database environment. Consider the following:

- Read access to data in a linked vault will be somewhat slower than that in a copied vault, but write transactions will pay the performance penalty when copied vaults are used.
- If the communications network is down between servers, you can still access data in a copied vault, although no one on either side of the connection will be able to modify it. Vaults that your server accesses via a link will not be accessible to you; however, users connected to its master server will have both read and write access.
- Some vaults and ingested stores may not need to be distributed at all.
- Administrative data must be accessible at all times to all servers.

The sections that follow provide more guidelines on the available distribution methods.

When to Replicate the Admin Vault

Given a reliable Wide Area Network (WAN) connection with adequate capacity between server locations, the best performance is achieved by replicating the Admin vault to all sites. Not all servers in each site need to have a copy of the Admin vault, but if each site has only one server, then all servers should get a copy of it.

Every time a definition is added or modified, all copies of the Admin vault must be available from the point of origin for the transaction, or the transaction will not be committed. For this reason, avoid distributing the Admin vault until the development phase is complete. The rate of modification will be greatly reduced once the eMatrix application is in production, and therefore the advantages provided by replicating the Admin data will then outweigh the disadvantage of the need for total database availability to make Admin updates.

When to Link Remote Servers to a Central Admin Vault

If the database connection is less reliable between the server locations, it may be advisable to leave the Admin vault in the master location, and use database links to serve the Admin definitions to the remote sites. If the network link is down to one site, that site will not be able to access the eMatrix application, even though the local server may contain the vault for most of that site's business objects, and the store for its files. Because network interruptions can interfere with the update of admin definitions if the Admin vault is replicated, during the development phase for multiple site implementations, database links are recommended.

When to Replicate Data Vaults

Data vaults should be copied to remote sites when the data they contain needs to be accessed frequently in read and write mode from more than one location. If the business objects in a particular vault are commonly connected to business objects in vaults mastered in other locations, it is a good candidate for replication. Users will be best served by this approach because they will get the complete information as they navigate from object to object, in spite of temporary network interruptions. Replication will actually reduce the WAN traffic produced by eMatrix clients because clients will be getting their data from their LAN's replica of the vault.

When to Link Remote Servers to a Master Server's Vault

Data vaults should be linked but not copied when the data in that vault applies primarily to only its host location, though it may be accessed by clients anywhere in the enterprise. When disk consumption is a concern, using links makes the most sense, as it does not require redundant disk consumption at other servers. A very large vault, mastered on a large server, may not fit on some of the smaller servers in the distributed database network. Links will rely on the availability of the WAN connection and will load the WAN pipe more heavily with eMatrix traffic, as information must be retrieved from servers at other locations.

Using Schema Names

eMatrix must use database links as a means of connecting different Oracle users when the users are in different Oracle instances. But since it is very common to setup a distributed, replicated, loosely-coupled, or open federation all within a single Oracle instance, *schema names* can be used instead, which greatly improves performance. Note this is intended primarily for setting up a demo environment, but it could certainly be used in production, as long as the requirements listed below can be met.

Requirements

To use schema names, there are two requirements:

- First, the Oracle user to which you are connected must have rights to select, insert, update, and remove data in the other schemas. These rights are established in the Oracle Security Manager. One simple way to enable this is to make the eMatrix user a DBA, which has these rights on all schemas. Refer to Oracle documentation for the procedure for changing user rights.
- Also, the environment variable, MX_USE_SCHEMA_NAMES must be set to “true” in the initialization file. Until this setting is made, the system will continue to use database links.

Use Case

Consider the following example:

eMatrix Server Name	User Name	Password	Connect string
Server1	user1	xxxx	taurus
Server2	user2	xxxx	taurus

Both these Server objects are referencing tables in the same Oracle instance. By default, eMatrix will create a database link called *Server2* in the *Server1* schema and another database link called *Server1* in the *Server2* schema, and reference all table names through the database links. For instance, when an eMatrix user connected to *Server1* requests the list of business types stored on *Server2*, the following SQL command is issued when database links are in use:

```
Select * from mxbusstype@Server2;
```

With schema names, instead of using a database link, the system prefixes the table name with the instance user name. The following SQL command is equivalent to the above query using schema names:

```
Select * from user2.mxbusstype;
```

The significant difference is that the transaction is local. A duplicate connection to *taurus* is not needed, and the request does not have to go “there and back again.”

8

Working with Indices

Index Defined	186
Defining an Index.....	187
Description Clause.....	187
Icon Clause.....	187
Attribute Clause	188
Hidden Clause	188
Property Clause.....	188
Modifying an Index Definition.....	189
Enabling/Disabling an Index.....	190
Improving Performance with Indices.....	191
Validating an Index	191
Using the index as select output.....	191
Deleting an Index	192

Index Defined

eMatrix stores attributes in separate rows in the database, allowing flexible and dynamic business modeling capabilities. This has the side effect of requiring extra SQL calls (joins) when performing complex queries and expands that specify multiple attribute values. To improve performance of these operations, you can define an *index*. In fact, test cases show marked improvement on many types of queries and expands.

An *index* is a collection of attributes that when enabled, causes a new, single-row table to be created for each vault. The columns in the table represent unique attribute values. If a commonly executed query uses multiple attributes as search criteria, you should consider defining an index. Once enabled, searches involving the indexed attributes generate SQL that references the index table instead of the old attribute tables, eliminating the need for a join.

When an index is created or attributes are added or removed from it, the index is by default disabled. The new database tables are created and populated when the index is enabled. This step can be time-consuming; roughly 30 seconds for 100,000 objects. However, it is assumed that an index will be enabled by a System Administrator once when created or modified, and will remain enabled for normal system operation.

Indexing attributes is only one way of optimizing performance. A properly tuned database is critical. While the *eMatrix Installation Guide* and *eMatrix MQL Guide* provide some guidelines and procedures, refer to your database documentation for tuning details.

Write operations of indexed attributes occur in 2 tables instead of just 1 and so a performance penalty is paid. Because of this, disabling indices prior to performing bulk load/update operations is recommended.

Defining an Index

An index is created with the add index statement:

```
add index NAME [ADD_ITEM {ADD_ITEM}];
```

NAME is the name you assign to the index. The index name is limited to 127 characters. For additional information, refer to *Business Object Name* in Chapter 34.

ADD_ITEM is an add index clause which provides additional information about the index:

description VALUE
icon FILENAME
attribute NAME{, NAME}
[! not]hidden
property NAME [to ADMINTYPE NAME] [value STRING]

For example:

```
add index CostAndWeight  
    description "Actual Costs and Weight"  
    icon index.gif  
    attribute "Actual Cost", "Actual Weight";
```

When you create an index, the mxindex table is updated. The lx tables are not created until the index is enabled.

Each clause is described below.

Description Clause

The Description clause of the Add Index statement provides general information for you about the function of the index you are defining. For example:

```
Attributes for Finance Query  
Attributes for Project Status Report  
Attributes for Cost Analysis
```

There is no limit to the number of characters you can include in the description.

Icon Clause

Icons help users locate and recognize items. You can assign a special icon to the new index or use the default icon. The default icon is used when in view-by-icon mode of eMatrix System Modeler. Any special icon you assign is used when in view-by-image mode of eMatrix System Modeler. When assigning a unique icon, you must use a .gif image file. Refer to *Icon Clause* in Chapter 1 for a complete description.

.gif filenames should not include the @ sign, as that is used internally by eMatrix.

Attribute Clause

The Attribute clause of the Add Index statement assigns attributes to the index. These attributes must be previously defined with the `add attribute` statement. (See [Defining an Attribute](#) in Chapter 12.) If they are not defined, an error message is displayed.

A maximum of 20 attributes can be placed in a single index. A typical index will have between 2 and 4 attributes. It is recommended not to include multiline or at least string attributes that will typically hold more than 255 characters.

Generally, all attributes placed in an index should exist in the same type definitions. For example, if an index is created with the attributes Cost, Weight, and Quantity, then all relationships or types that reference Cost, Weight, or Quantity should reference all three of them. If there is a type or relationship that references only some of the attributes in an index, a warning is generated. You can proceed with the index creation, but it will take longer to create an index where this completeness test fails.

Hidden Clause

You can specify that the new index is “hidden” so that it does not appear in the index chooser. You may want to use the hidden option if, for example, an object is under development or if it is intended only for your personal use. Users who are aware of the hidden index’s existence can enter its name manually where appropriate. Hidden objects are also accessible through MQL.

Property Clause

Integrators can assign ad hoc attributes, called Properties, to the index. Properties allow associations to exist between administrative definitions that aren’t already associated. The property information can include a name, an arbitrary string value, and a reference to another administration object. The property name is always required. The value string and object reference are both optional. The property name can be reused for different object references, that is, the name joined with the object reference must be unique for any object that has properties.

```
add index NAME  
    property NAME [to ADMINTYPE NAME] [value STRING];
```

For additional information on properties, see *Overview of Administration Properties* in Chapter 24.

Modifying an Index Definition

After an index is defined, you can change the definition with the modify index statement. This statement lets you add or remove defining clauses and change the value of clause arguments:

```
modify index NAME [MOD_ITEM {MOD_ITEM}];
```

NAME is the name of the index you want to modify.

MOD_ITEM is the modification you want to make.

You can make the following modifications. Each is specified in a Modify Index clause, as listed in the following table. Note that you need to specify only fields to be modified.

Modify index Clause	Specifies that...
name NEW_NAME	The current index name changes to that of the new name entered.
description VALUE	The current description, if any, changes to the value entered.
icon FILENAME	The image is changed to the new image in the file specified.
add attribute NAME	The named attribute is added to the index's list of attributes.
remove attribute NAME	The named attribute is removed from the index's list of attributes.
hidden	The hidden option is changed to specify that the object is hidden.
[! not]hidden	The hidden option is changed to specify that the object is not hidden.
property NAME [to ADMINTYPE NAME] [value STRING]	The named property is modified.
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.

Enabling/Disabling an Index

An index must be enabled before it is used to improve performance. The following command can be executed from MQL provided that the current user is defined as a System Administrator.

```
enable index NAME;
```

As soon as you enable the index, the ixXXXXXX_XXXXX table gets created in every vault. If you then disable the index, or add or remove attributes, these tables are “closed.” To use the index it must be re-enabled. Closed tables are removed when the system is shut down.

For example, to enable the index named “Index for Finance queries,” enter the following MQL statement:

```
enable index "Index for Finance queries";
```

To disable a defined index, use:

```
disable index NAME;
```

Improving Performance with Indices

Validating an Index

Since up-to-date database statistics are vital for optimal query performance, after enabling an index you should generate and add statistics to the new database tables. Use the following command to do so:

```
validate level 4 index NAME [output FILENAME]
```

The output FILENAME clause is used for DB2 only.

This is assuming statistics are already up-to-date for all other tables. Refer to [Chapter 3, Validating the eMatrix Database](#) for more information.

Using the Index as Select Output

You can use the `index[]` selectable on business objects and relationships to retrieve the attribute values directly from the index table. This is roughly N times faster than using `attribute[]` where N is the number of attributes in the index. For example, if you routinely run a query/select like the following:

```
temp query bus Part * * select attribute\[Part_Number\]  
attribute\[Quantity\] attribute\[AsRequired\]  
attribute\[Process_Code\] attribute\[Function_Code\]  
attribute\[Safety_Indicator\] dump | output d:/partlist.txt;
```

You could create an index called Parts containing all the attributes listed and use the following query instead:

```
temp query bus Part * * select index\[Parts\] dump | output d:/  
partlist.txt;
```

The temp query that uses the select index is roughly 6 times faster than using the query that selects all 6 attributes separately.

Selecting an index is similar to selecting attributes with a few differences:

- `Select index` returns values for all attributes in the index, where `select attribute` returns just one. If an attribute does not apply to a type or relationship, eMatrix returns null.
- Strings longer than 255 characters will be truncated to 255 characters. Generally these are defined as multiline attributes, which should not be included in an index.

Deleting an Index

If an index is no longer required, you can delete it with the Delete Index statement:

```
delete index NAME;
```

NAME is the name of the index to be deleted.

When this statement is processed, eMatrix searches the list of indices. If the name is not found, an error message is displayed. If the name is found, the index is deleted.

For example, to delete the index named “Index for Finance queries,” enter the following MQL statement:

```
delete index "Index for Finance queries";
```

Working With Export/Import

Overview of Export/Import.....	195
Exporting Administrative Objects.....	196
Export statement.....	196
ADMIN_TYPE Clause.....	196
OPTION_ITEMS.....	197
XML Clause	198
Into File/Onto File	198
Exclude Clause.....	198
Log FILENAME Clause.....	198
Exception FILENAME Clause.....	199
Exporting Business Objects.....	200
Export Bus Statement.....	200
OPTION_ITEMS.....	201
Exporting Workflows.....	203
Export Workflow Statement	203
Exporting Objects to XML.....	204
XML Export Requirements and Options	204
XML Clause	205
XML Statement.....	205
XML Output.....	205
Importing Administrative Objects.....	209
Import Statement	209
List Clause	210
Admin and ADMIN_TYPE Clauses.....	210
OPTION_ITEMS.....	211
Use FILE_TYPE Clauses	212
Importing Servers	213
Importing Workspaces	213
Importing Properties	214
Importing Business Objects.....	216
Import Bus Statement.....	216
List Clause	217
OPTION_ITEMS.....	217
Use FILE_TYPE Clauses	219
Importing Workflows	221
Import Workflow Statement	221
Extracting from Export Files.....	222
OPTION_ITEMS.....	222

Examples	222
Migrating Databases	223
Migrating Files	223
Comparing Schema	225
Scenario	225
Creating a Baseline	225
Comparing a Schema with the Baseline	226
Reading the Report	229
Examples	232

Overview of Export/Import

Administrative definitions, business object metadata and workflows, as well as checked-in files, can be exported from one eMatrix root database and imported into another. Exporting and importing can be used across eMatrix schemas of the same eMatrix version level, allowing definitions and structures to be created and “fine tuned” on a test system before integrating them into a production database.

When you export a business object, you can save the data to a *eMatrix Exchange* file or to an XML format file that follows the eMatrix.dtd specification. An eMatrix Exchange file is created according to the eMatrix Exchange Format. This format or the XML format must be adhered to strictly in order to be able to import the file. For more information on the XML format, see [Exporting Objects to XML](#).

The eMatrix Exchange Format is subject to change with each eMatrix version.

Exporting Administrative Objects

Before exporting any objects, it is important to verify that MQL is in native language mode. Exporting in the context of a non-native language is not supported. If you need to redefine the language, use the `push alias` command, as described in [Overriding the Language](#) in Chapter 25.

Export statement

Use the Export statement to export the definitions of an eMatrix database. The complete export command for administrative objects is as follows:

```
export ADMIN_TYPE TYPE_PATTERN [OPTION_ITEM [OPTION_ITEM]...] | into | file FILE_NAME  
| onto |  
[exclude EXCLUDE_FILE] [log LOG_FILE] [exception EX_FILE];
```

`ADMIN_TYPE` is the administrative definition type to be exported. It can be any of the following:

admin	group	person	role	type
association	index	policy	rule	user
attribute	inquiry	process	server	vault
command	location	program	site	wizard
form	menu	relationship	store	workflow
format	page	report	table	

The `ADMIN_TYPE` is followed by a `TYPE_PATTERN` which filters the definitions to be exported.

`OPTION_ITEM` is an export clause which further defines the requirements of the export to be performed. It can be any of the following:

!archive	incremental [N]	xml
continue	!mail	.
!icon	!set	.

`!mail` and `!set` (`notmail` and `notset`) are only meaningful when exporting person objects but should not cause syntax errors for other types.

`OPTION_ITEMS` can be used in any order before the `into | onto file` clause.

`FILENAME` is the name of a new or existing file to which to write the export data.

Each clause is described in the sections that follow.

ADMIN_TYPE Clause

The `ADMIN_TYPE` clause of the Export statement is used to specify which administrative definitions to export. Valid values are shown in the table above. Specifying `admin` will export all administrative types of the name or pattern that follows. For example, you may

have been testing a schema and have created definitions for attributes, types, policies, relationships, and so on—all beginning with “TEST.” To export all the test administrative objects you might use the following:

```
export admin TEST* into file testdefs.mix;
```

To export all administrative objects, you could use the following:

```
export admin * into file alladmin.mix;
```

*Using export admin * does not export creator and guest user accounts.*

Instances of specific administrative object types can be exported using the appropriate administrative object form of this command. For example, to export all policies you might use:

```
export policy * into file policy.mix;
```

OPTION_ITEMS

!Icon Clause

When exporting administrative objects, the associated icons are included by default. By using the `!icon` clause in the Export statement, the icon can be omitted. For example, to export all programs into one file and omit the associated icons, use:

```
export program * !icon into file programs.mix;
```

!Mail and !Set Clauses

When exporting person objects, there is an associated workspace. The workspace contains IconMail and sets (as well as Visuals) saved by the person being exported. When exporting Persons, workspaces are exported by default, but you can omit mail and sets using the `!mail` and `!set` clauses. These clauses are valid with the `export person` and `export admin` statements only. Refer to the discussion in [Migrating Databases](#) for information on why you would want to exclude these items.

`!mail` and `!set` (not`mail` and not`set`) are only meaningful when exporting Person objects but should not cause syntax errors for other types.

For example, to export all persons without mail or sets, the following can be used:

```
export person * !mail !set into file person.mix;
```

Incremental N Clause

Use the `incremental` clause to export a specified number (N) of unarchived objects. Unarchived objects are objects that have not yet been exported, or have been changed since they were last exported. If no number is specified, all unarchived objects are exported. Without this clause, export does not look at the `archived` setting and exports all objects fitting the criteria.

Note that while it is possible to export administrative changes incrementally, it is recommended that during database migration, Business and System Administration tasks are avoided.

!Archive Clause

The `!archive` clause is used to tell eMatrix not to set the archive setting on objects being exported. This is used to facilitate performance of large exports that will not require an incremental export.

Continue Clause

The `continue` clause tells eMatrix to proceed with additional exports even if an error is generated. When `continue` is used, it is helpful to use the `log` and/or `exception` clauses as well, so that diagnostics can be performed, and the data that caused the error is trapped.

XML Clause

Use the XML clause to export data into XML format. For details, see [Exporting Objects to XML](#).

Into File/Onto File

Exported data can be appended to a file or written to a new file by using the `onto file` or `into file` forms of the Export statement. The `into file` form creates a new file, or overwrites an existing file of the name specified as `FILENAME`. The `onto file` form appends to an existing file.

Exclude Clause

Use the Exclude clause to point to a map file that lists any objects to be excluded. It includes the `exclude` keyword and a map file name. For example:

```
export admin TEST* into file teststuf.mix exclude nogood.txt;
```

The contents of the exclude map file for Administrative objects must follow the following format:

```
ADMIN_TYPE NAME  
ADMIN_TYPE NAME
```

`ADMIN_TYPE` can be any of the administrative object types: vault, store, location, server, attribute, program, type, relationship, format, role, group, person, policy, report, form, association, rule, workflow, or process.

`NAME` is the name of the definition instance that should be excluded in the import. Wildcard patterns are allowed.

As indicated, each definition to be excluded must be delimited by a carriage return. The `exclude` clause is optional.

Log FILENAME Clause

Apply this clause if you want to specify a file to contain error messages and details for the export process. The output is similar to using the `verbose` flag, but includes more details.

Exception FILENAME Clause

Exception FILENAME provides a file location where objects are written if they fail to export. The file will contain the type and name of any objects that could not be exported. This is often used with a log file so that after the diagnostics are performed from the information in the log file, the exception file can be used as a guide to know what should be exported.

Exporting Business Objects

Before exporting any objects, it is important to verify that MQL is in native language mode. Exporting in the context of a non-native language is not supported. If you need to redefine the language, use the `push alias` command, as described in [Overriding the Language](#) in Chapter 25.

Export Bus Statement

Use the Export Businessobject statement to export business objects from a vault or a set:

```
export bus[inessobject] OBJECTID [from |vault VAULT_NAME|] [OPTION_ITEM {OPTION_ITEM}]  
|set SET_NAME  
|into | file FILENAME [FILE_TYPE FILENAME [FILE_TYPE FILENAME]...];  
|onto |
```

OBJECTID is the OID or Type Name Revision of the business object. It may also include the `in VAULTNAME` clause, to narrow down the search. Wildcard patterns are allowed.

VAULT_NAME is the name of the vault from which to export the business object(s). While neither is required, you can specify either a vault or a set from which to export, but not both.

SET_NAME is the name of the set from which to export the business objects. While neither is required, you can specify either a vault or a set from which to export, but not both.

OPTION_ITEM is an export clause which further defines the requirements of the export. It can be any of the following:

!archive	!history	!state
!captured	!icon	xml
continue	incremental [N]	.
!file	!relationship	.

OPTION_ITEMS can be used in any order before the `into|onto file` clause.

FILENAME is the name of the file in which to store the exported information.

FILE_TYPE FILENAME offers a way to log errors and trap exceptions, as well as to exclude objects in the export. FILE_TYPE can be any of the following:

exclude	log	exception
---------	-----	-----------

Note that when FILE_TYPES are specified on export, the use keyword is not required. However, when used on import, it is.

Excluding Information

When exporting business objects, the default is to include everything about the object. However, the **OPTION_ITEMS** can be used to further define the requirements of the export to be performed, by specifying information to exclude. The following clauses can be used to exclude information when exporting business objects.

<code>!captured</code>	<code>!icon</code>	<code>!relationship</code>
<code>!file</code>	<code>!history</code>	<code>!state</code>

For example, to export a single object without including history, use:

```
export businessobject Assembly "ABC 123" A !history into file  
obj.mix;
```

To export a single object without including its icon, use:

```
export businessobject Assembly "ABC 123" A !icon into file  
obj.mix;
```

To export a single object without any of its relationships, use:

```
export businessobject Assembly "ABC 123" A !relationship into  
file obj.mix;
```

To export all objects from vault TEMP and reset the current state to the beginning of the lifecycle, use:

```
export businessobject * * * from vault TEMP !state into file  
obj.mix;
```

OPTION_ITEMS can be used in any order before the `into|onto file` clause. Refer to the descriptions of the other **OPTION_ITEMS** in the [OPTION_ITEMS](#) section of [Exporting Administrative Objects](#).

Exclude Clause

Use the **Exclude** clause to point to a file that lists any objects to be excluded. For example:

```
export businessobject TEST* into file teststuf.mix exclude nogood.txt;
```

The contents of the exclude file for business objects must follow the following format:

```
businessobject OBJECTID  
businessobject OBJECTID
```

`OBJECTID` is the OID or Type Name Revision of the business object. It may also include the `in VAULTNAME` clause, to narrow down the search.

As indicated, each business object to be excluded must be delimited by a carriage return.

Excluding Files

When exporting (or importing) business objects, the options for its files are:

1. `!file`, which tells eMatrix not to export files when exporting business objects. By default, both file data and content are exported.
2. `!captured`, which tells eMatrix not to export captured store file content. This flag affects content only, not file metadata.

3. Otherwise, both file metadata and actual file contents are written to the export file. In this case any file sharing is lost (resulting in duplication of files).

When `!file` is used, the `captured` flag cannot be used, since you cannot include files without metadata. (Attempts to do this will not return an error, but no file information or content will be included). Refer to [Migrating Databases](#) for more information on file migration strategies.

Exporting Workflows

Before exporting any objects, it is important to verify that MQL is in native language mode. Exporting in the context of a non-native language is not supported. If you need to redefine the language, use the `push alias` command, as described in [Overriding the Language](#) in Chapter 25.

Export Workflow Statement

Use the Export Workflow statement to export workflows from a vault or a set:

```
export workflow PROCESS_PATTERN [from vault VAULT_NAME] [OPTION_ITEM [OPTION_ITEM]...]
| into | file FILENAME [FILE_TYPE FILENAME [FILE_TYPE FILENAME]...];
| onto |
```

`PROCESS_PATTERN` includes both the name of the process on which the workflow is based and the workflow name. Wildcard patterns are allowed. For example, to export the “Assembly 4318” workflow, which is based on the process “Create Part,” you would use the following:

```
export workflow "Create Part" "Assembly 4318";
```

`VAULT_NAME` is the name of the vault from which to export the workflow(s).

`OPTION_ITEM` is an export clause which further defines the requirements of the export. It can be any of the following:

!archive	continue	incremental [N]
!history	!icon	xml

`OPTION_ITEMS` can be used in any order before the `into | onto file` clause.

`OPTION_ITEMS` are similar to those used when exporting business objects. See [OPTION_ITEMS](#) in the [Exporting Business Objects](#) section for details.

`FILENAME` is the name of the file in which to store the exported information.

`FILE_TYPE FILENAME` offers a way to log errors and trap exceptions, as well as to exclude objects in the export. `FILE_TYPE` can be any of the following:

exclude	log	exception
---------	-----	-----------

Note that when FILE_TYPES are specified on export, the use keyword is not required. However, when used on import, it is.

Exporting Objects to XML

The eXtensible Markup Language (XML) standard, defined by the World Wide Web Consortium (W3C), is becoming increasingly important as an Internet data exchange medium. Like HyperText Markup Language (HTML), XML is a text-based tag language based on Standard Generalized Markup Language (SGML). Unique features that distinguish XML from HTML include:

- Whereas HTML is a Web presentation language, XML is a data description language that defines the structure of data rather than how it is presented.
- Whereas HTML uses a fixed set of tags and attributes, XML lets authors define their own tags and attributes, providing complete control over the structure of data in an XML document. This makes XML “extensible” and “self-describing,” as tag names can incorporate domain-specific terminology.
- Whereas HTML combines presentation markup with content markup, XML separates presentation from content. By accessing presentation information stored in XSL or CSS style sheets, the same XML data can be presented in different formats on different devices.

These features give XML great flexibility as a standard way of exchanging structured data across platforms and applications and displaying that data in a variety of web-enabled devices (e.g., computers, cell phones, PDAs).

For eMatrix users, XML can enhance business-to-business EDI functions by facilitating data exchange and application integration with a partner’s ERP or other data-driven systems. XML will make it easy for you to import eMatrix data from customers and partners and display that data in a variety of formats. You will also be able to read an XML export files into any version of eMatrix, assuring backward as well as forward compatibility among versions of eMatrix software.

XML Export Requirements and Options

eMatrix schema and business objects may optionally be exported in XML format. The resulting XML stream follows rules defined in the eMatriXML Document Type Definition (DTD) file (named “ematrixxml.dtd”) that is installed with eMatrix in the /XML directory. This DTD file is referenced by all exported eMatrix XML files. In order to interpret and validate eMatrix XML export files, an XML parser must be able to access the eMatriXML DTD file. This means that the eMatriXML DTD file should reside in the same directory as the exported XML files and be transferred along with those files to any other user or application that needs to read them.

Using MQL, you can export eMatrix objects to XML format in either of two ways:

- Insert an XML clause in any Export statement for exporting administrative, business, or workflow objects
- Issue an XML statement to turn on XML mode as a global session setting. In this mode, any Export statements you enter, with or without the XML clause, will automatically export data in XML format.

Be aware of the following restrictions related to exporting and importing in XML format:

- When exporting programs to XML, make sure the code does not contain the characters “[>”. These characters can be included as long as there is a space between the bracket and the greater sign. For more information, see *Programs Exported to XML* in Chapter 19.

- Legal characters in XML are the tab, carriage return, line feed, and the legal graphic characters of Unicode, that is, #x9, #xA, #xD, and #x20 and above (HEX). Therefore, other characters, such as those created with the ESC key, should not be used for ANY field in eMatrix, including business and administrative object names, description fields, program object code, or page object content.

XML Clause

Use the XML clause as an OPTION_ITEM in any Export statement to export administrative, business, or workflow objects in XML format. To ensure that the XML file(s) reside in the same directory as the eMatriXML DTD, you can export files to the XML folder (in your MATRIXHOME directory) where the eMatriXML DTD is installed. Alternatively, you can specify another location in your Export statement and copy the DTD into that directory. Use the `into` clause in your Export statement and specify a full directory path, including file name. For example:

```
export person guy xml into file c:\matrix\xml\person.xml;
```

XML Statement

Use the XML statement to turn XML mode on or off as a global session setting. The complete XML command syntax is:

```
xml [on|off];
```

Omitting the on/off switch causes XML mode to be toggled on or off, depending on its current state. XML mode is off by default when you start an MQL session.

For example, to export a person named “guy” using an XML statement along with an Export statement, you can enter:

```
xml on;
export person guy into file c:\matrix\xml\person.xml;
```

If you need to export several eMatrix objects to XML format, using the XML statement to turn on XML mode first can eliminate the need to re-enter the XML clause in each Export statement as well as the possibility of an error if you forget.

XML Output

The XML export format typically generates a file 2 to 3 times larger than the standard eMatrix export format. To conserve space, subelements are indented only if verbose mode is turned on. Indentation makes output more readable but is not required by an XML parser. Some XML viewers (like Internet Explorer 5.0) will generate the indentation as the file is parsed.

The following example shows standard export output when you export a person named “guy” during an MQL session. While relatively compact, this output is not very intelligible to a user.

```
MQL<1>set context user creator;
MQL<2>export person guy;
!MTRX!AD! person guy 8.0.2.0
guy Guy ""
    "" "" "" 0 0
    1 1 1 0 1 0 1 "" "" *
    1111101111111100111110
    0 1 .finder * GuyzViewTest 0 * FileInDefaultFormat 1 ""
    0 0
    0 0
    0
    0
    0
    de3IJEE/JIJJ.
    ""
    0
    0
    0
    1
    Test ""
    1
    Description description 1
    0 0 0 0 70 21 0 0 1 1
    1
    0 0
    0
    0 0
person guy successfully exported.
!MTRX!END
export successfully completed
```

The next example shows XML output when you use the Export statement, with XML mode turned on, to export a person named “guy” during a continuation of the same MQL session. While more intelligible to a user, this code creates a larger output file than the standard eMatrix export format.

```
MQL<3>xml on;
MQL<4>verbose on;
MQL<5>export person guy;
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!-- (c)MatrixOne, Inc., 2000 -->
<!DOCTYPE ematrix SYSTEM "ematrixxml.dtd">
<ematrix>
    <creationProperties>
        <release>8.0.2.0</release>
        <datetime>2000-05-05T17:57:19Z</datetime>
        <event>export</event>
        <dtdInfo>&ematrixProductDtd;</dtdInfo>
    </creationProperties>
    <person id="0.1.31721.46453">
        <adminProperties>
            <name>guy</name>
        </adminProperties>
        <fullName>Guy</fullName>
        <fullUser/>
```

```

<businessAdministrator/>
<systemAdministrator/>
<applicationsOnly/>
<passwordChangeRequired/>
<access>
  <all/>
</access>
<adminAccess>
  <attributeDefAccess/>
  <typeAccess/>
  <relationshipDefAccess/>
  <formatAccess/>
  <personAccess/>
  <roleAccess/>
  <associationAccess/>
  <policyAccess/>
  <programAccess/>
  <wizardAccess/>
  <reportAccess/>
  <formAccess/>
  <ruleAccess/>
  <siteAccess/>
  <storeAccess/>
  <vaultAccess/>
  <serverAccess/>
  <locationAccess/>
</adminAccess>
<queryList count="1">
  <query>
    <name>.finder</name>
    <queryStatement>
      <vaultPattern>*</vaultPattern>
      <typePattern>GuyzViewTest</typePattern>
      <ownerPattern>*</ownerPattern>
      <namePattern>FileInDefaultFormat</namePattern>
      <revisionPattern>1</revisionPattern>
    </queryStatement>
  </query>
</queryList>
<password>de3IJEE/JIJJ.</password>
<tableList count="1">
  <table>
    <name>Test</name>
    <columnList count="1">
      <column>
        <label>Description</label>
        <expression>description</expression>
        <usesBusinessObject/>
        <geometry>
          <xLocation>0.0</xLocation>
          <yLocation>0.0</yLocation>
          <width>70.0</width>
          <height>21.0</height>
          <minWidth>0.0</minWidth>
          <minHeight>0.0</minHeight>
          <autoWidth/>
          <autoHeight/>
        </geometry>
        <editable/>
      </column>
    </columnList>
  </table>
</tableList>

```

```
</column>
</columnList>
</table>
</tableList>
</person>
</eMatrix>
```

Importing Administrative Objects

When migrating objects or entire databases, it is important to import in the correct order:

1. Import all administrative objects first.
2. Import all business objects.
3. Import workspaces from the same exported ASCII data file as was used to import Persons. Refer to [Importing Workspaces](#) for more information.
4. Import workflows. Refer to [Importing Workflows](#) for more information.
5. Import properties of the administrative objects that have them. The same file that was used to import the administrative objects should be used. Refer to [Importing Properties](#) for more information.

For more migration strategies refer to [Migrating Databases](#).

Import Statement

Use the Import statement to import administrative objects from an export file to an eMatrix database. The export file must be in eMatrix Exchange Format or in an XML format that follows the eMatrix.dtd specification.

```
import [list] [property] ADMIN_TYPE TYPE_PATTERN [OPTION_ITEM [OPTION_ITEM]...]
from file FILENAME [use [FILE_TYPE FILE [FILE_TYPE FILE]]...];
```

ADMIN_TYPE is the administrative definition type to be imported. It can be any of the following:

admin	group	person	role	type
association	index	policy	rule	user
attribute	inquiry	process	server	vault
command	location	program	site	wizard
form	menu	relationship	store	workflow
format	page	report	table	

The ADMIN_TYPE may include a TYPE_PATTERN which filters the definitions to be imported.

OPTION_ITEM is an import clause that further defines the requirements of the import to be performed. It can be any of the following:

```
!icon           !overwrite        continue
commit N       skip N          pause N
```

OPTION_ITEMS can be used in any order before the from file clause. The sections below describe these options.

FILENAME is the path and name of the existing .mix file from which to import the information.

`FILE_TYPE` can be used to specify files to be used to control the import and log files and exceptions. `FILE_TYPE` can be any of the following:

map	exclude	log	exception
-----	---------	-----	-----------

Note that the `use` keyword must be used with any files that are specified: map files, exclude files, log files, or exception files. If more than one file type is to be used, the `use` keyword should only be stated once. (The `use` keyword is not used at all in the export command).

`FILE` is the name of the file to be used with `FILE_TYPE`. For example, an exclude `FILE` would be the name of the existing file that lists any eMatrix objects which should not be included in the import.

List Clause

The List clause of the Import statement is used to show on the screen what would be done on import. It enables you to perform a practice run, ensuring that the files to be imported are read correctly. For example:

```
import list admin * from file c:\admin.mix;
```

This might output:

```
vault Standards
vault ...
store Drawings
store ...
attribute Sheet Count
attribute ...
program Cadra
program ...
type Drawing
type ...
relationship Report
relationship ...
format Cadra
format ...
role Manager
role ...
group Sales
group ...
person Dave
person ...
policy Reports
policy ...
```

Admin and ADMIN_TYPE Clauses

The Admin clause of the Import statement is used to import all administrative types of the name or pattern specified (`TYPE_PATTERN`). For example, you may have exported to one file a test schema with definitions for attributes, types, policies, relationships, and so on—all beginning with “TEST”. To import all the test administrative types, you might use the following:

```
import admin TEST* from file TestStuff.mix
```

Instances of all administrative object types can be imported using the specific administrative object with this command as well. For example, to import all policies in an XML export file, you might use:

```
import policy * from file policy.xml;
```

OPTION_ITEMS

!Icon Clause

By default, the associated icon is imported. By using the `!icon` clause in the Import statement, the icon can be omitted. For example, to import the program EngTable from one file and omit the associated icon, use:

```
import program EngTable !icon from file programs.mix;
```

If the icon was omitted during the export, it cannot be included during import, regardless of the use of this clause.

Overwrite Clause

The `overwrite` clause is used if objects were incrementally exported. If an object already exists, the overwrite clause tells the import command to modify any objects that are found to already exist. Without the overwrite clause, if an administrative object exists, the import will fail, since it will try to create a new object with the same name as an existing object.

When `!overwrite` is used, the objects that don't import because they already exist are written to the exception file (if specified).

When importing, the default for administrative objects is `!overwrite`. For business objects, the default is `overwrite`.

Overwrite will not work on administrative objects that are referenced by business objects. This eliminates the possibility of doing things like `import overwrite` to change icon images, or editing an administrative object in a development environment, testing, and then importing into a production environment.

Skip N Clause

`Skip N` provides a way not to import the beginning `N` number of objects in an export file. It is helpful if a previous attempt to import a file was unsuccessful and aborted part way through the process.

Commit N Clause

The `commit` clause gives you the ability to specify `N` number of objects to enclose in transaction boundaries. In this way, some progress could be made on the import even if some transactions are aborted. The default is 100.

Pause N Clause

Pause allocates the amount of time in seconds to wait between import transactions. It is used with the `commit` clause. A pause is beneficial when import is running in the background for an extended period of time. It provides a larger window of opportunity for you to access the database machine's resources. Import transactions run continuously when this clause is not included.

Continue Clause

The `continue` clause tells eMatrix to proceed with additional imports even if an error is generated. When `continue` is used, it is helpful to use the `log` and/or `exception` clauses as well, so that diagnostics can be performed, and the data that caused the error is trapped.

Use FILE_TYPE Clauses

You can indicate that a file should be generated or referenced with the `use FILE_TYPE` clause. Note that the `use` keyword must be included with any clauses that specify map files, exclude files, log files, or exception files. If more than one file type is listed, the `use` keyword should only be stated once. (The `use` keyword is not used at all with the `export` command.) `FILE_TYPE` can be any of the following:

Map Clause

The Map clause of the Import statement indicates a map file which lists any new locations or names for objects. The map file must use the following format:

```
ADMIN_TYPE OLDDNAME NEWNAME  
ADMIN_TYPE OLDDNAME NEWNAME
```

`ADMIN_TYPE` can be any of the administrative object types: vault, store, location, server, attribute, etc.

`OLDDNAME` is the name of the instance that will be found in the import file.

`NEWNAME` is the name that should be substituted for `OLDDNAME` when imported into the new database.

As indicated, each definition to be changed must be delimited by a carriage return.

Exclude Clause

Use the Exclude clause of the Import statement to indicate a file that lists any objects to be excluded. For example:

```
import admin TEST* from file teststuf.mix use exclude nogood.obj;
```

The exclude file for administrative objects must use the following format:

```
ADMIN_TYPE NAME  
ADMIN_TYPE NAME
```

`ADMIN_TYPE` can be any of the administrative object types: vault, store, location, server, attribute, program, type, relationship, format, role, group, person, workspace, policy, report, form, association, rule, workflow, or process.

NAME is the name of the definition instance that should be excluded in the import. Wildcard patterns are allowed.

As indicated, each definition to be excluded must be delimited by a carriage return.

Log FILE Clause

Apply the `log FILE` clause to specify a file to contain error messages and details for the import process. The output is similar to using the verbose flag, but includes more details.

Exception FILE Clause

Use the `exception FILE` clause to provide a file location for objects to be written to if they fail to import. If a transaction aborts, all objects from the beginning of that transaction up to and including the “bad” object will be written to the exception file.

Importing Servers

If a server object is imported (via MQL or Oracle) into a different Oracle instance or user than it was exported from, the username and password settings of the server must be modified before distributed access is available. This is particularly important if an upgrade will follow the import, since all servers must be accessible to the machine doing the upgrade.

Importing Workspaces

Workspaces contain a user’s queries, sets, and iconmail, as well as all Visuals. Workspaces are always exported with the person they are associated with. However, when Persons are imported, the workspace objects are not included. This is because they may rely on the existence of other objects, such as Types and business objects, which may not yet exist in the new database. Workspaces must be imported from the same .mix file that was used to import persons. For example:

```
import workspace * from file admin.mix;
```

Or:

```
import workspace julie from file person.mix;
```

Importing Workflows

To import a specific workflow, you must specify the process name followed by the workflow name after the keyword `workflow`:

```
import workflow PROCESS_NAME WORKFLOW_NAME;
```

For example, to import the “Assembly 4318” workflow, which is based on the process “Create Part,” you would use the following:

```
import workflow "Create Part" "Assembly 4318";
```

Wildcards can be used. For example, to import all workflows, use * for the process and workflow names:

```
import workflow * *;
```

If you are importing an entire database, you must include workflows. For example:

```
import admin *
import bus * * *
import workflow * *
import workspace *;
```

These import statements must be performed in this order. Workflows can refer to business objects, so they must be done after `import bus`. Also, TaskMails require that the appropriate person objects exist, so `import person` or `import admin` must precede `import workflow`.

Importing TaskMail with Workflows

eMatrix users can neither create nor delete TaskMails. They simply receive TaskMails and work on them. Therefore, exporting and then importing them in a static manner could end up generating TaskMails for those tasks that may have already been completed. For this reason, during the import of the workflow, TaskMail is handled as follows:

- If a workflow being imported is completed or stopped, the workflow import does not generate any TaskMail.
- If a workflow is active or suspended, the workflow import generates TaskMail if the TaskMail is missing from the person's in-basket for those tasks that are active. This happens when a workflow is imported to an empty database.
- If the workflow is active or suspended, TaskMail is not re-generated during import if the active tasks are already available in the person's in-basket. This situation may occur when a workflow is imported in overwrite mode.
- When importing with `overwrite`, if the workflow in the database is suspended and the workflow in the export file is active, the import would do all that is necessary to make the workflow consistent with the workflow in the export file. In this case, some TaskMails may be taken away, and others added, in order to bring the database to the proper state.

Importing Properties

Properties are sometimes created to link administrative objects to one another. Like workspaces, properties are always exported with the administrative object they are on. Import is somewhat different, however, since a property may have a reference to another administrative object, and there is no way to ensure that the referenced object exists in the new database (administrative objects are sometimes exported and then imported in pieces). So a command to import administrative objects is issued, the specified objects are created first, and then the system attempts to import its properties. If the "continue" modifier is used, the system will get all the data it can, including system and user properties. But to ensure that all properties are imported, even when the administrative data may have been contained in several files, use the `import property` command.

For example, data can be exported from one database as follows:

```
export attrib * into file attrib.exp;
export program * into file program.exp;
export type * into file type.exp;
export person * into file person.exp;
```

Then the administrative objects are imported:

```
import attrib * from file attrib.exp;
import program * from file program.exp;
import type * from file type.exp;
import person * from file person.exp;
```

Finally, workspaces and any “missed” properties are imported. Note that properties may exist on workspace objects, so it is best to import properties after workspaces:

```
import workspace * from file person.exp;
import property attrib * from file attrib.exp;
import property program * from file program.exp;
import property type * from file type.exp;
import property person * from file person.exp;
```

Importing Business Objects

If triggers are attached to an object being imported, they may be executed at the time of import. Therefore, the MQL command:

```
trigger off;
```

should be run before importing objects into a database.

Import Bus Statement

Use the Import Businessobject statement to import business objects from an export file to an eMatrix database. The export file must be in eMatrix Exchange Format or in an XML format that follows the eMatrix.dtd specification.

```
import [list] bus[inessobject] OBJECTID [OPTION_ITEM [OPTION_ITEM]...]
from file FILENAME [use [FILE_TYPE FILE [FILE_TYPE FILE]]];
```

OBJECTID is the OID or Type Name Revision of the business object. It may also include the in VAULTNAME clause, to narrow down the search. Wildcard patterns are allowed.

OPTION_ITEM is an import clause which further defines the requirements of the import to be performed. It can be any of the following:

[!]attribute	continue	[!]overwrite	pause N
[!]basic	[!]file	[!]relationship	skip N
[!]captured	[!]history	!fromrelationship	[!]preserve
commit N	[!]icon	!torelationship	[!]state
from vault VAULT_NAME to vault VAULT_NAME			

OPTION_ITEMS can be used in any order before the from FILE clause. The sections below describe these options.

FILENAME is the name of the file from which to get the exported ASCII data.

FILE_TYPE can be used to specify files to be used to control the import and log files and exceptions. FILE_TYPE can be any of the following:

map	exclude	log	exception
-----	---------	-----	-----------

Note that the use keyword must be used with any files that are specified: map files, exclude files, log files, or exception files. If more than one file type is to be used, the use keyword should only be stated once. (The use keyword is not used at all in the export command).

FILE is the name of the file to be used with FILE_TYPE. For example, an exclude FILE would be the name of the existing file that lists any eMatrix objects which should not be included in the import.

List Clause

The List clause of the Import statement is used to show on the screen what would be done on import. It enables you to perform a practice run, ensuring that the files to be imported are read correctly. For example:

```
import list businessobject * * * from file c:\revb.mix;
```

This might output:

```
businessobject Drawing 568872 B  
businessobject Drawing ERC 7144 B  
businessobject ...
```

OPTION_ITEMS

From Vault and To Vault Clauses

One way to redirect the import of business objects into a new location (vault) is to use the `from vault` clause with the `to vault` clause. For example, to place all business objects that were in vault Test into vault Prod in the new database, use:

```
import businessobject * * * from vault Test to vault Prod from  
file c:\revb.mix;
```

Another alternative for redirecting business objects during import is to use a map file as discussed in the section [Use Map Clause](#).

Excluding Information

When importing business objects, the default is to include everything about the object. However, you may specify that some parts of the .mix file should not be imported.

Any information that was omitted during the export cannot be included during import, regardless of the use of this clause.

The table below shows the options which can be used when importing business objects to exclude information:

Clause	Used to:
<code>!attribute</code>	Exclude attribute values. Generally used with the overwrite option, so that even though other parts of the object will be overwritten, attribute values will not be.
<code>!basic</code>	Exclude basic information. Generally used with the overwrite option, so that even though other parts of the object will be overwritten, basic information will not be.
<code>!file</code>	Exclude file metadata and content of captured store files. See Excluding Files for more information.
<code>!captured</code>	Exclude file content but include metadata of captured store files. See Excluding Files for more information.
<code>!history</code>	Exclude history entries
<code>!icon</code>	Exclude icons.
<code>!relationship</code>	Exclude all relationships.
<code>!torelationship</code>	Exclude "to" relationships.
<code>!fromrelationship</code>	Exclude "from" relationships.
<code>!state</code>	Exclude state information. Generally used with the overwrite option, so that even though other parts of the object will be overwritten, current state and signature information will not be. This can also be used to put objects back to the first state in their lifecycle.

For example, to import all objects from file mystuff.mix and exclude all checked in files, use:

```
import businessobject * * * !file from file mystuff.mix;
```

For captured files, file metadata can be included, without the actual file content by using the `!captured` option. See the discussion [Excluding Files](#) for more information.

To import a single object without including history, use:

```
import businessobject Assembly "ABC 123" A !history from file
obj.mix;
```

To import a single object without any of its relationships, use:

```
import businessobject Assembly "ABC 123" A !relationship from
file obj.mix;
```

When importing objects, several options are available with regard to relationship information:

- Include all relationship information (default)

- Exclude all relationship information (!relationship)
- Exclude *to* relationships (!torelationship)
- Exclude *from* relationships (!fromrelationship).

To import all objects and reset the current state to the beginning of the lifecycle, use:

```
import businessobject * * * !state from file obj.mix;
```

!Overwrite Clause

!overwrite tells import not to replace any objects that already exist with the new object information from the specified .mix file. Specifying **!overwrite** will cause an error if an object already exists. Default behavior for business objects is to overwrite files.

*When importing, the default for administrative objects is **!overwrite**. For business objects, the default is **overwrite**.*

Preserve Clause

The Preserve clause is used when importing business objects. It specifies not to change the object's modification date to the date of the import.

See specific sections under [Importing Administrative Objects](#) for clauses that are also available when importing business objects.

Use FILE_TYPE Clauses

Use Map Clause

Another way to change the location of objects upon import is with the **use map** clause. A map file, which lists any new locations for objects, is specified. The map file must use the following format:

```
vault OLDNAME NEWNAME
```

OLDNAME is the name of the Vault that will be found in the import file.

NEWNAME is the name of the Vault that the object is imported into in the new database.

If multiple vaults are listed, they must be delimited by a carriage return.

For example, to import and place everything from vault Test to vault Prod using a map file, the map file entry would be: vault Test Prod and the command would be:

```
import businessobject * * * from file c:\stuff.mix use map  
c:\newvault.map
```

Changing vaults on import is also possible using the [From Vault and To Vault Clauses](#).

Exclude Clause

Use the Exclude clause to indicate a file that lists any objects to be excluded. For example:

```
import businessobject TEST* from file teststuf.mix use map changes.map exclude  
nogood.obj;
```

The exclude map file for business objects must use the following format:

```
businessobject OBJECTID  
businessobject OBJECTID
```

OBJECTID is the OID or Type Name Revision of the business object. It may also include the `in VAULTNAME` clause, to narrow down the search.

As indicated, each business object to be excluded must be delimited by a carriage return.

Import Example

The following command:

- imports business objects from export.mix
- commits the import transaction every 5 business objects
- writes the audit trail to import.log
- writes exception objects to error.mix

```
import bus * * * commit 5 continue from file export.mix use log  
import.log exception error.mix;
```

If an error occurs, you can look at the entries in the import.log to see what went wrong. Once the problem is resolved, the error.mix file could be used to import the objects that were not imported successfully the first time.

When object imports fail, the transaction aborts and rolls back the import of any objects that precede it in the transaction boundary. These are the objects written to the exception file. A new transaction is started with the next object. Using the example above, suppose the third object attempted caused a problem; the transaction is aborted. The first three objects are written to error.mix and a new transaction begins with the fourth object. After the eighth object was imported, the transaction is committed and the new database has five new objects.

Importing Workflows

Import Workflow Statement

Use the Import Workflow statement to import workflows from an export file to an eMatrix database.

```
import [list] workflow PROCESS_PATTERN [OPTION_ITEM [OPTION_ITEM]...]
from file FILENAME [use [FILE_TYPE FILE [FILE_TYPE FILE]]...];
```

PROCESS_PATTERN includes both the name of the process on which the workflow is based and the workflow name. Wildcard patterns are allowed. For example, to import the “Assembly 4318” workflow, which is based on the process “Create Part,” you would use the following:

```
import workflow "Create Part" "Assembly 4318";
```

OPTION_ITEM is an import clause which further defines the requirements of the import to be performed. It can be any of the following:

[!]attribute	commit N	[!]history	[!]overwrite	pause N
[!]basic	continue	[!]icon	[!]relationship	skip N
from vault VAULT_NAME to vault VAULT_NAME				

OPTION_ITEMS can be used in any order before the from FILE clause. These OPTION_ITEMS are similar to those used for importing business objects. For details, see the items detailed in the [Importing Business Objects](#) section.

VAULT_NAME is the name of the vault you are importing from and to.

FILENAME is the name of the file from which to get the exported ASCII data.

FILE_TYPE can be used to specify files to be used to control the import and log files and exceptions. FILE_TYPE can be any of the following:

map	exclude	log	exception
-----	---------	-----	-----------

Note that the use keyword must be used with any files that are specified: map files, exclude files, log files, or exception files. If more than one file type is to be used, the use keyword should only be stated once. (The use keyword is not used at all in the export command).

FILE is the name of the file to be used with FILE_TYPE. For example, an exclude FILE would be the name of the existing file that lists any eMatrix workflows that should not be included in the import.

Extracting from Export Files

Sometimes export files contain more information than you want imported. When this is the case, the extract statement can be used to create a new file containing only the specified information of the original file.

```
extract |bus OBJECTID      | [OPTION_ITEM [OPTION_ITEM]]from file FILENAME |into| file NEW;  
       |ADMIN ADMIN_NAME|                                |onto|
```

OBJECTID is the OID or Type Name Revision of the business object. It may also include the in VAULTNAME clause, to narrow down the search. If a pattern is listed, the first match is extracted.

ADMIN is any of the administrative types to be extracted.

ADMIN_NAME is the name of the administrative object to be extracted.

OPTION_ITEM can be any of the following:

remaining	skip N	exclude FILE
-----------	--------	--------------

FILENAME is an existing export file from which to extract information.

NEW is the file that extract creates or appends with the requested information.

OPTION_ITEMS

Remaining Clause

The remaining clause of the Extract statement tells eMatrix to extract from the specified object to the end of the file.

Skip Clause

The skip clause is used to begin the extraction at a place other than the beginning of the file. N is the number of beginning entry marks (!MTRX) to skip.

Exclude FILE Clause

The exclude FILE clause is used to point to a FILE that lists any objects to be excluded from the extraction. This exclude file must use the same format as for the [Exclude Clause](#) when exporting business objects.

Examples

For example, if all administrative objects were extracted into one file called admin.mix, you can extract all policies into a separate file as follows:

```
extract policy * from file admin.mix into file policy.mix;
```

To extract all business object from the fifth entry until the end of the file use:

```
extract bus * * * skip 4 remaining from file objects.mix into file newobjs.mix;
```

Migrating Databases

When migrating entire databases or a large number of objects, it is important to import in the following order:

1. Import all administrative objects first. For example:

```
import admin * from file admin.mix;
```

2. Import all business objects. For example:

```
import bus * * * from file bus.mix;
```

3. Import all workflows. For example:

```
import workflow * * from file wkfl.mix;
```

4. Import workspaces from the same exported ASCII data file as was used to import Persons. For example:

```
import workspace * from file admin.mix;
```

Or:

```
import workspace julie from file person.mix;
```

Workflows can refer to business objects, so they must be done after `import bus`. TaskMail is imported by `import workspace` and refers to workflows. So `import workspace` must be done last.

Workspaces contain a user's queries, sets, TaskMail, and IconMail, as well as all Visuals. Workspaces are always exported with the person with which they are associated. However, when Persons are imported, the workspace objects are not included. This is because they may rely on the existence of other objects, such as Types and business objects, which may not yet exist in the new database.

5. Import properties of administrative objects.

When administrative objects are imported, eMatrix imports all the objects without their properties and then goes back and imports both system and user properties for those objects. If administrative objects were exported in pieces, such as all Types in one file, all Persons in another, then properties should be explicitly imported with the `import property` command. Refer to [Importing Properties](#) for more information.

Migrating Files

When migrating business objects, there are three options for its files:

1. By default, both file metadata and actual file contents are written to the export file. eMatrix UUencodes the file and writes it to the export data file, along with business object metadata. Pointers to the files are guaranteed because the file is recreated during import. In this case, any file sharing is lost, (as when revisions use the same file list—each revision in the chain gets its own copy of the file).
2. Adding the `!file` clause tells eMatrix not to include any file metadata or content.
3. Using the `!captured` clause tells eMatrix not to include captured store file content. This option writes only the fully qualified path of checked in files in the data file, along with business object metadata.

When migrating databases, most often the `!captured` option is recommended. This will facilitate the process, in that the `.mix` files will not be as large, and therefore will not require as much disk space or processing time to complete the migration. Once the import is complete, the objects will point to the appropriate files in the same location. The same is true for tracked files; if metadata is included for tracked files, the imported objects will point to the appropriate location, as long as the store names and hosts have stayed the same. For ingested files, the options are to include both metadata and content or not to include either.

The key to keeping the file pointers accurate is keeping the store path definitions consistent. For example, let's say the database from which we are exporting has a captured store named "Released Data Store." The path of this store is defined as "/company/released." To maintain pointer consistency when using `!captured`, the new database must also have a defined captured store "Released Data Store" with the same path definition. If stores are exported and then imported, there is no problem. However, if stores are first created in the new database, to redistribute them onto different machines, for example, problems could occur with objects that have directories checked in. Since the export function needs to UUencode the directory structure and files, the machine defined as the host in the original store definition must still be accessible to MQL through NFS.

If objects are to be deleted and then re-imported, use the `!captured` option, but be sure to tar off any captured store directories before deleting any objects. Once the objects are deleted, the directories restored, and the objects imported, the files will be associated with the appropriate objects.

Migrating Revision Chains

Files may be shared among revisions of a business object. This "file sharing" concept was introduced to minimize storage requirements and is primarily used with captured stores. However, this does mean special attention is required when exporting and importing. Consider the following:

If the entire revision chain is not going to be exported and/or imported into a new database, then the use of `!captured` may result in lost files. For example, if the business object Assembly 123 0 has three files checked in, and is then revised to Assembly 123 1, this new revision shares the three files with the original. As long as both are exported and imported, `!captured` can be used (in fact, should be used to avoid file duplicating). However, if just Assembly 123 1 is imported into a new database, then all file data should be imported. If `!captured` was used instead, the imported business object Assembly 123 1 would not have any files!

Comparing Schema

This section describes how to compare two schemas to determine the differences between them. A *schema* is all the administrative objects needed to support an eMatrix application. Use schema comparison to compare:

- Different versions of the same schema to manage changes to the schema.
- Two schemas from different databases so you can merge the schemas (for example, merge a checkout system with a production system).

The process of comparing schema involves two main steps:

1. Create a baseline sample of one schema using XML export. See [Creating a Baseline](#).
2. Analyze the differences between the baseline sample and the other schema (or a later version of the same schema) using the compare command. You specify the administrative types (attributes, relationships, etc.) and a name pattern (for example, all attributes beginning with “Supplier”) to compare. Each compare command outputs a single log file that contains a report. The report lists the administrative objects in the baseline export file that have been added, deleted, and modified. See [Comparing a Schema with the Baseline](#).

If your goal is to merge the two schemas by making the necessary changes to one of the schemas (sometimes called “applying a delta”), you can make the changes manually or by writing an MQL script file that applies the changes to the target schema.

Scenario

Suppose you need to determine the changes that have occurred in a checkout database versus what continues to exist in the production database. In this case, you may want to create a baseline of both databases, and use each to compare against the other. One report would be useful to find out what has changed in the checkout database. The other report would be useful to determine what it would take to apply those changes to the production database.

Creating a Baseline

The first step for comparing two schemas is to establish a baseline for analysis by sampling one of the schemas. You create a baseline by exporting administrative objects to XML format. You can use any option available for the export command to create the baseline export files. For information on options and more details about the export command, see [Exporting Administrative Objects](#).

Use the following guidelines to perform the export.

- Start MQL using a bootstrap file that points to the database containing the schema for which you want to create the baseline.
- There are two ways to produce an XML export file: toggle on XML mode and issue a normal export command, or issue a normal export command but include the XML keyword. For example:

```
xml;  
export ADMIN_TYPE TYPE_PATTERN into file FILENAME;
```

Or the equivalent:

```
export ADMIN_TYPE TYPE_PATTERN xml into file FILENAME;
```

- It's best to create separate export files for each administrative type and to keep all the objects of a type in one file. For example, export all attributes to file attributes.xml, all relationships to relationship.xml, etc. This keeps the baseline files to a reasonable size, and also lets you compare specific administration types, which makes it easy to produce separate reports for each administration type. If you need to identify subsets of objects within an export file to focus the analysis, you can do so using the compare command.
- The compare command requires that the ematrixxml.dtd file be in the same directory as the export files. Therefore, you should create the export files in the directory that contains the dtd file or copy the dtd file into the directory that contains the export files. If you don't specify a path for the export file, eMatrix creates the file in the directory that contains mql.exe file.

The following table shows examples of export commands that export different sets of administrative objects. All the examples assume the XML mode is not toggled on and that the ematrixxml.dtd file is in the directory d:\eMatrix\xml.

To export:	Use this command
all attributes	export attribute * xml into file d:\eMatrix\xml\attributes.xml
all attributes that begin with the prefix "Supply"	export attribute Supply* xml into file d:\eMatrix\xml\SupplyAttributes.xml
all administrative objects that begin with the prefix "mx" (usually better to keep all objects of a type in separate files)	export admin mx* xml into file d:\eMatrix\xml\mxApp.xml

Comparing a Schema with the Baseline

After creating the baseline for one of the schemas, the second step is to analyze the differences between the baseline and the second schema, and generate a report that lists the differences. The MQL compare command performs this step. The syntax for the compare command is shown below. Each clause and option in the command is explained in the following sections.

```
compare ADMIN_TYPE TYPE_PATTERN [workspace] from file FILENAME [use [map FILENAME]
[exclude FILENAME] [log FILENAME] [exception FILENAME]];
```

When issuing the command, make sure you start MQL using a bootstrap file that points to the database that you want to compare with the schema for which you created the baseline.

The compare command analyzes only administrative objects and ignores any business objects and workflow instances in the baseline export file.

ADMIN_TYPE TYPE_PATTERN Clause

The ADMIN_TYPE clause specifies which administrative types to compare. Valid values are:

admin	group	person	role	type
association	index	policy	rule	user
attribute	inquiry	process	server	vault
command	location	program	site	wizard
form	menu	relationship	store	workflow
format	page	report	table	

The value admin compares all administrative types of the name or pattern that follows. For example, the clause “admin New*” compares all administrative objects with the prefix “New.” All other ADMIN_TYPE values compare specific administrative types. For example, to compare all policies, you could use:

```
compare policy * from file d:\eMatrix\xml\policy.xml;
```

To compare objects of a particular type whose names match a character pattern, include the pattern after the ADMIN_TYPE clause. For example, to compare only relationships that have the prefix “Customer”, you could use:

```
compare relationship Customer* from file d:\eMatrix\xml\relationship.xml;
```

workspace Option

The workspace option applies only when comparing administrative types that can have associated workspace objects: persons, groups, roles, or associations. When you add the keyword “workspace” to the compare command, any workspace objects (tips, filters, cues, toolsets, sets, tables, and views) owned by the persons, groups, roles, or associations being compared are included in the comparison operation.

For example, suppose you compare the Software Engineer role and the only change for the role is that a filter has been added. If you don’t use the workspace option, the compare operation will find no changes because workspace objects aren’t included in the comparison. If you use the workspace option, the comparison operation will report that the role has changed and the filter has been added.

from file FILENAME Clause

The from file FILENAME specifies the path and name of the existing XML export file. This file contains the baseline objects you want to compare with objects from the current schema. The ematrixxml.dtd file (or a copy of it) must be in the same directory as the baseline XML file.

use FILETYPE FILENAME option

Use the use keyword once for any optional files specified in the compare command: map files, exclude files, log files, or exception files. If more than one file type is to be used, the use keyword should be stated once only.

The use FILETYPE FILENAME option lets you specify optional files to be used in the compare operation. FILETYPE accepts four values:

- log

The use log FILENAME option creates a file that contains the report for the compare operation. The compare command can be issued without identifying a log file, in which case just a summary of the analysis is returned in the MQL window—total number of objects analyzed, changed, added, deleted. The log file report lists exactly which objects were analyzed and describes the differences. More information appears in the report if verbose mode is turned on. For more information about the report, see [Reading the Report](#).

An efficient approach is to run the compare command without a log file to see if any changes have occurred. If changes have occurred, you could turn on verbose mode, re-run the compare command and supply a log file to capture the changes.

- map

A map file is a text file that lists administrative objects that have been renamed since the baseline file was created. The map file maps names found in the given baseline file with those found in the database (where renaming has taken place). Use the map file option to prevent the compare operation from finding a lot of changes simply because administrative objects had their names changed. The map file must use the following format:

```
ADMIN_TYPE OLDDNAME NEWNAME  
ADMIN_TYPE OLDDNAME NEWNAME
```

OLDDNAME is the name of the object in the baseline export file. NEWNAME is the name of the object in the current schema (the schema you are comparing against the baseline file). Include quotes around the names if the names include spaces. Make sure you press Enter after each NEWNAME so each renamed object is on a separate line (press Enter even if only one object is listed). For example, if the Originator attribute was renamed to Creator, the map file would contain this line:

```
attribute Originator Creator
```

If no map file is specified, the compare operation assumes that any renamed objects are completely new and that the original objects in the baseline were deleted.

- exclude

An exclude file is a text file that lists administrative objects that should not be included in the comparison. The exclude file must use the following format:

```
ADMIN_TYPE NAME  
ADMIN_TYPE NAME
```

NAME is the name of the administrative object that should be excluded in the compare operation. Make sure you press Enter after each NAME so each excluded object is on a separate line (press Enter even if only one object is listed). Wildcard patterns are

allowed. Include quotes around the names if the names include spaces. For example, if you don't want to compare the Administration Manager role, the exclude file would contain this line:

```
role "Administration Manager"
```

- exception

The use exception FILENAME option creates a file that lists objects that could not be compared. If a transaction aborts, all objects from the beginning of that transaction up to and including the “bad” object are written to the exception file.

FILENAME is the path and name of the file to be created (log and exception files) or used (map and exclude files). If you don't specify a path, eMatrix uses the directory that contains the mql.exe file.

Reading the Report

If you specify a log file in the compare command, the compare operation generates a report that lists all objects analyzed and the changes. The report format is simple ASCII text. The report contains enough information to enable an expert eMatrix user to write MQL scripts that apply the changes to a database.

Below is a sample of a report with the main sections of the report indicated. Each section of the report is described below.

Preamble	A map file was not given. An exclude file was not given. Input baseline file: 'd:\eMatrix\xml\person1.xml'. Type = 'person', Name Pattern = 'Joe C*', Workspace 'included'. Start comparison 'Wed Jun 21, 2000 3:57:09 PM EDT' Baseline version was '9.0.0.0'. Current version is '9.0.0.0'. =====
Banner for each object analyzed	===== 'person' 'Joe Consultant' ===== ===== 'person' 'Joe Chief_Engineer' =====
Banner for each sub-object analyzed	----- 'query' 'ECR's in Process' ----- ----- 'set' 'Products' -----
Change analysis section that describes changes	businessObjectRef objectType 'Assembly Work Instruction' objectName 'WI-300356' objectRevision 'D' has been deleted. businessObjectRef objectType 'Assembly Work Instruction' objectName 'WI-300356' objectRevision 'C' has been added.
Summary	End comparison 'Wed Jun 21, 2000 3:57:11 PM EDT' 0 objects have been added. 0 objects have been deleted. 1 objects have been changed. 4 objects are the same.

Preamble—Lists the clauses and options used in the compare command, the time of the operation, and software version numbers.

Object banner—Each object analyzed is introduced with a banner that includes the object type and name wrapped by “=” characters.

Sub-object banner—Each sub-object analyzed for an object is listed under the object banner. The sub-object banner includes the sub-object type and name wrapped by “_” characters. In the above example, only one object, Joe Chief_Engineer, has sub-objects, which in this case is a query and a set.

Change Analysis—Following the banner for each object and sub-object analyzed, there are four possibilities:

1. If no changes are found, then no analysis lines appear. The next line is the banner for the next object/sub-object or the summary section.
2. If the object (sub-object) has been added, then the following line appears: “Has been added.”
3. If the object (sub-object) has been deleted, then the following line appears: “Has been deleted.”
4. If the object (sub-object) has been changed, there are three possibilities:
 - a) If a field has been added, then the following line appears: “FIELD has been added.”
 - b) If a field has been deleted, then the following line appears: “FIELD has been deleted.”
 - c) If a field has changed, then the following line appears: “FIELD has been changed.”

where FIELD is in the following form: FIELDTYPE [‘FIELDNAME’]
 [SUBFIELDTYPE [‘FIELDNAME’]] ...

and FIELDTYPE identifies the type of field using tags found in the ematrixxml.dtd file, and FIELDNAME identifies the name of the field when more than one choice exists.

The best way to identify the field that has changed is to traverse the XML tree structure, looking at element names (tags) and the value of any name elements (placed in single quotes) along the way. Use the ematrixxml.dtd file as a roadmap. Element names never have single quotes around them, and values of name elements always have single quotes around them. This should help parsing logic distinguish between the two.

Here are some sample messages that would appear in the change analysis section if the compare operation finds that an object has changed (possibility 4):

```
frame 'Change Class' has been added.
typeRefList 'Change Notice' has been deleted.
field fieldType 'select' has been deleted.
widget 'ReasonForChange' multiline has been changed
widget 'ReasonForChange' validateProgram programRef has been changed.
businessObjectRef objectType 'Assembly Work Instruction' objectName 'WI-300356'
objectRevision 'D' has been deleted.
```

Summary—The final section of the report contains a timestamp followed by the same summary that appears in the MQL window.

Verbose Mode

Turn on verbose mode to see more details in the report for changed objects/sub-objects (possibility 4 in the above description). To see these additional details, make sure you turn on verbose mode before issuing the compare command.

Verbose mode does not produce additional information if an object has been added (possibility 2 in the above description) or deleted (possibility 3). To get more information about an added object, use a print command for the object. To gather information about a deleted object, look at the XML export file used for the baseline.

When the operation finds changes to objects, verbose mode adds text as follows:

- For possibility 4a (field has been added), the keyword “new” appears followed by VALUE.
- For possibility 4b (field was deleted), the keyword “was” appears followed by VALUE.
- For possibility 4c (field was changed), the keywords “was” and “now” appear, each followed by VALUE.
where VALUE is either the actual value (in single quotes) or a series of name/value pairs (where the value portion of the name/value pair is in single quotes).

Here are some sample messages that would appear in the change analysis section if the compare operation finds that an object has changed (possibility 4) and verbose mode is turned on:

```
field fieldType 'select' has been added.
new absoluteX '0' absoluteY '0' xLocation
'382.500000' yLocation '36.000000' width
'122.400002' height '24.000000'
autoWidth '0' autoHeight '0' border '0'
foregroundColor 'red' backgroundColor ''
fieldValue 'name' fontName 'Arial Rounded MT
Bold-10' multiline '0' editable '0'

field fieldType 'select' has been deleted.
was absoluteX '0' absoluteY '0' xLocation
'382.500000' yLocation '36.000000' width
'122.400002' height '24.000000'
autoWidth '0' autoHeight '0' border '0'
foregroundColor 'red' backgroundColor ''
fieldValue 'name' fontName 'Arial Rounded MT
Bold-10'

width has been changed.
was '795.0'
now '792.0'

field fieldType 'label' has been changed.
was absoluteX '0' absoluteY '0' xLocation
'191.250000' yLocation '110.000000' width
'252.449997' height '24.000000'
autoWidth '0' autoHeight '0' border '0'
foregroundColor '' backgroundColor ''
fieldValue 'Maximum Distance Between Centers:
fontName ''
now absoluteX '0' absoluteY '0' xLocation
'191.250000' yLocation '108.000000' width
'252.449997' height '24.000000'
autoWidth '0' autoHeight '0' border '0'
foregroundColor '' backgroundColor ''
fieldValue 'Maximum Distance Between Centers:
fontName ''
```

Comparing Person objects

eMatrix does not include the default users creator and guest when you export all person objects with:

```
MQL< >export person * xml into file /temp/person.xml;
```

So if you then compare this exported file to another schema, even if the same Person objects exist, the compare output will show that 2 objects have been added. For example:

```
MQL<7>compare person * from file /temp/person.xml use log  
person.log;  
2 objects have been added.  
0 objects have been removed.  
0 objects have been changed.  
172 objects are the same.
```

The log will show

```
....  
===== 'person' 'guest' =====  
Has been added.  
===== 'person' 'creator' =====  
Has been added.
```

Examples

Line <2> places the session into XML mode. All subsequent export commands will generate export files in XML format.

Line <3> exports all programs (including wizards) that start with "A".

Line <4> exports all persons.

Below are two example MQL sessions. The first MQL session shows two baseline export files being created.

Matrix Query Language Interface, Version 9.0.0.0

Copyright (c) 1993-2000 MatrixOne, Inc.

All rights reserved.

```
MQL<1>set context user Administrator;  
MQL<2>xml on;  
MQL<3>export program A* into file d:\eMatrix\xml\program1.xml;  
MQL<4>export person * into file d:\eMatrix\xml\person1.xml;  
MQL<5>quit;
```

The second MQL session shows several comparisons being performed using the baseline export file person1.xml. It is assumed changes have occurred in the database, or the session is being performed on a different database.

Matrix Query Language Interface, Version 9.0.0.0

Copyright (c) 1993-2000 MatrixOne, Inc.

All rights reserved.

MQL<1>set context user Administrator;

MQL<2>compare person "Joe C*" from file d:\eMatrix\xml\person1.xml;

0 objects have been added.

0 objects have been removed.

0 objects have been changed.

5 objects are the same.

MQL<3>compare person "Joe C*" workspace from file d:\eMatrix\xml\person1.xml;

0 objects have been added.

0 objects have been removed.

1 objects have been changed.

4 objects are the same.

MQL<4>verbose on;

MQL<5>compare person "Joe C*" workspace from file d:\eMatrix\xml\person1.xml use log d:\eMatrix\xml\person1w.log;

0 objects have been added.

0 objects have been removed.

1 objects have been changed.

4 objects are the same.

compare successfully completed.

This portion of the MQL window shows a continuation of the previous session. Here, the baseline export file program1.xml is used for several more comparisons. The sections that follow show the contents of the files used in the compare commands.

MQL<6>verbose off;

MQL<7>compare program A* from file d:\eMatrix\xml\program1.xml use log d:\eMatrix\xml\program1.log map d:\eMatrix\xml\program1.map;

2 objects have been added.

0 objects have been removed.

3 objects have been changed.

11 objects are the same.

compare successfully completed.

MQL<8>verbose on;

MQL<9>compare program A* from file d:\eMatrix\xml\program1.xml use log d:\eMatrix\xml\program2.log map d:\eMatrix\xml\program1.map;

2 objects have been added.

0 objects have been removed.

3 objects have been changed.

11 objects are the same.

compare successfully completed.

MQL<10>compare program A* from file d:\eMatrix\xml\program1.xml use log program3.log map d:\eMatrix\xml\program1.map exclude d:\eMatrix\xml\program1.exc;

2 objects have been added.

0 objects have been removed.

2 objects have been changed.

10 objects are the same.

compare successfully completed.

Line <2> compares all person objects that start with "Joe C" with the baseline file person1.xml. Since no log file is specified, no report is generated. (Not having a log file would typically be done to see if anything has changed.) The summary message states that none of the 5 objects analyzed have changed.

Line <3> performs the same compare but also includes workspace items assigned to the persons. The results now show that there has been a change. To view the changes, a report must be generated.

Line <4> turns on verbose mode.

Line <5> performs the previous compare but also gives a log file to place the report into. The resulting report can be found in [Reading the Report](#).

Line <7> performs a compare on all program objects that start with the letter "A" but also includes a map file that identifies a rename of one of the program objects.

See [program1.map](#) and [program1.log](#). Notice that "use" is only used once even though two files are used (a log file and a map file).

Line <9> performs the same compare as in Line <7> but with verbose mode turned on. Look at the two reports ([program1.map](#) and [program2.log](#)) to see the difference between verbose off and on.

Line <10> performs the same compare but adds an exclude file that eliminates two program objects from the analysis (thus leading to two fewer objects mentioned in the results). See [program1.exc](#) and [program3.log](#).

program1.map

```
program "Add Task" "Add Task Import"
```

program1.log

```
Map file 'd:\eMatrix\xml\program1.map' successfully read.  
An exclude file was not given.  
Input baseline file: 'd:\eMatrix\xml\program1.xml'.  
Type = 'program', Name Pattern = 'A*', Workspace 'excluded'.  
Start comparison at 'Wed Jun 21, 2000 2:13:49 PM EDT'.  
Baseline version was '9.0.0.0'.  
Current version is '9.0.0.0'.  
=====  
===== 'program' 'Add Component (As-Designed)' =====  
description has been changed.  
===== 'program' 'Add Assembly (As-Designed)' =====  
===== 'program' 'Add Purchase Requisition' =====  
===== 'program' 'Add Event' =====  
===== 'program' 'AttributeProg' =====  
===== 'program' 'A' =====  
===== 'program' 'A1' =====  
===== 'program' 'A2' =====  
===== 'program' 'Add ECR' =====  
----- 'frame' 'Master Frame' -----  
----- 'frame' 'Welcome' -----  
----- 'frame' 'Change Type' -----  
----- 'frame' 'Reason For Change' -----  
width has been changed.  
widget 'ReasonForChange' multiline has been changed.  
widget 'ReasonForChange' validateProgram programRef has been changed.  
widget 'postECR Input1label4' fontName has been changed.  
widget 'Reason For Changelabel5' widgetValue has been changed.  
----- 'frame' 'Product Line' -----  
widget 'Product Linelabel4' has been added.  
----- 'frame' 'Change Priority' -----  
----- 'frame' 'Reason for Urgency' -----  
widget 'Product Linelabel4' has been deleted.  
----- 'frame' 'AdditionalSignatures' -----  
----- 'frame' 'Conclusion' -----  
----- 'frame' 'Conclusion-Urgent' -----  
----- 'frame' 'Status Feedback' -----  
frame 'Change Class' has been added.  
===== 'program' 'Add Assembly' =====  
----- 'frame' 'Master Frame' -----  
----- 'frame' 'Welcome' -----  
----- 'frame' 'Make vs Buy' -----  
widget 'MakevsBuy' validateProgram programRef has been changed.  
----- 'frame' 'Part Family' -----  
----- 'frame' 'Assembly Description' -----  
----- 'frame' 'Target Parameters' -----  
----- 'frame' 'Status Feedback' -----  
'program' 'Add Task' mapped to 'Add Task Import'  
===== 'program' 'Add Task Import' =====  
----- 'frame' 'Master Frame' -----  
----- 'frame' 'Welcome' -----  
----- 'frame' 'Key Task Name' -----
```

```

----- 'frame' 'Task Description' -----
----- 'frame' 'Status Feedback' -----
===== 'program' 'Add Note' =====
----- 'frame' 'Master Frame' -----
----- 'frame' 'prepreStatus Feedback' -----
----- 'frame' 'preStatus Feedback' -----
===== 'program' 'Add Operation' =====
----- 'frame' 'Master Frame' -----
----- 'frame' 'Welcome' -----
----- 'frame' 'Key Operation Name' -----
----- 'frame' 'Operation Description' -----
----- 'frame' 'Status Feedback' -----
===== 'program' 'Add ECR Import' =====
----- 'frame' 'Master Frame' -----
----- 'frame' 'Welcome' -----
----- 'frame' 'Change Type' -----
----- 'frame' 'Change Class' -----
----- 'frame' 'Reason For Change' -----
----- 'frame' 'Product Line' -----
----- 'frame' 'Change Priority' -----
----- 'frame' 'Reason for Urgency' -----
----- 'frame' 'AdditionalSignatures' -----
----- 'frame' 'Conclusion' -----
----- 'frame' 'Conclusion-Urgent' -----
----- 'frame' 'Status Feedback' -----
===== 'program' 'Audiostream' =====
Has been added.
===== 'program' 'Add Task' =====
Has been added.
=====
End comparison at 'Wed Jun 21, 2000 2:13:51 PM EDT'.
2 objects have been added.
0 objects have been deleted.
3 objects have been changed.
11 objects are the same.

```

program2.log

```

Map file 'd:\eMatrix\xml\program1.map' successfully read.
An exclude file was not given.
Input baseline file: 'd:\eMatrix\xml\program1.xml'.
Type = 'program', Name Pattern = 'A*', Workspace 'excluded'.
Start comparison at 'Wed Jun 21, 2000 2:13:36 PM EDT'.
Baseline version was '9.0.0.0'.
Current version is '9.0.0.0'.
=====
===== 'program' 'Add Component (As-Designed)' =====
description has been changed.
was 'Matrix Prof Services: Contains settings for the program to create and connect and
a new object with AutoName logic.'
now 'Matrix Professional Services: Contains settings for the program to create and
connect and a new object with AutoName logic.'
===== 'program' 'Add Assembly (As-Designed)' =====
===== 'program' 'Add Purchase Requisition' =====
===== 'program' 'Add Event' =====
===== 'program' 'AttributeProg' =====
===== 'program' 'A' =====
===== 'program' 'A1' =====

```

```

===== 'program' 'A2' =====
===== 'program' 'Add ECR' =====
----- 'frame' 'Master Frame' -----
----- 'frame' 'Welcome' -----
----- 'frame' 'Change Type' -----
----- 'frame' 'Reason For Change' -----
width has been changed.
    was '380.0'
    now '360.0'
widget 'ReasonForChange' multiline has been changed.
    was '0'
    now '1'
widget 'ReasonForChange' validateProgram programRef has been changed.
    was 'NameCheck2'
    now 'NameCheck'
widget 'postECR Input1label4' fontName has been changed.
    was 'Arial-bold-14'
    now 'Arial-bold-10'
widget 'Reason For Changelabel5' widgetValue has been changed.
    was 'Enter The Stock Disposition:'
    now 'Enter Stock Disposition:'
----- 'frame' 'Product Line' -----
widget 'Product Linelabel4' has been added.
    new absoluteX '0' absoluteY '0' xLocation '198.900009' yLocation '72.000000' width
'160.650009' height '24.000000'
        autoWidth '0' autoHeight '0' border '0' foregroundColor '' backgroundColor ''
        widgetType 'label' widgetNumber '100002'
        widgetValue 'Enter Product Line' fontName 'Arial-bold-10'
----- 'frame' 'Change Priority' -----
----- 'frame' 'Reason for Urgency' -----
widget 'Product Linelabel4' has been deleted.
    was absoluteX '0' absoluteY '0' xLocation '198.900009' yLocation '72.000000' width
'160.650009' height '24.000000'
        autoWidth '0' autoHeight '0' border '0' foregroundColor '' backgroundColor ''
        widgetType 'label' widgetNumber '100002'
        widgetValue 'Enter Product Line' fontName 'Arial-bold-10'
----- 'frame' 'AdditionalSignatures' -----
----- 'frame' 'Conclusion' -----
----- 'frame' 'Conclusion-Urgent' -----
----- 'frame' 'Status Feedback' -----
frame 'Change Class' has been added.
===== 'program' 'Add Assembly' =====
----- 'frame' 'Master Frame' -----
----- 'frame' 'Welcome' -----
----- 'frame' 'Make vs Buy' -----
widget 'MakevsBuy' validateProgram programRef has been changed.
    was 'NameCheckTest'
    now 'NameCheck'
----- 'frame' 'Part Family' -----
----- 'frame' 'Assembly Description' -----
----- 'frame' 'Target Parameters' -----
----- 'frame' 'Status Feedback' -----
'program' 'Add Task' mapped to 'Add Task Import'
===== 'program' 'Add Task Import' =====
----- 'frame' 'Master Frame' -----
----- 'frame' 'Welcome' -----
----- 'frame' 'Key Task Name' -----
----- 'frame' 'Task Description' -----
----- 'frame' 'Status Feedback' -----

```

```

===== 'program' 'Add Note' =====
----- 'frame' 'Master Frame' -----
----- 'frame' 'prepreStatus Feedback' -----
----- 'frame' 'preStatus Feedback' -----
===== 'program' 'Add Operation' =====
----- 'frame' 'Master Frame' -----
----- 'frame' 'Welcome' -----
----- 'frame' 'Key Operation Name' -----
----- 'frame' 'Operation Description' -----
----- 'frame' 'Status Feedback' -----
===== 'program' 'Add ECR Import' =====
----- 'frame' 'Master Frame' -----
----- 'frame' 'Welcome' -----
----- 'frame' 'Change Type' -----
----- 'frame' 'Change Class' -----
----- 'frame' 'Reason For Change' -----
----- 'frame' 'Product Line' -----
----- 'frame' 'Change Priority' -----
----- 'frame' 'Reason for Urgency' -----
----- 'frame' 'AdditionalSignatures' -----
----- 'frame' 'Conclusion' -----
----- 'frame' 'Conclusion-Urgent' -----
----- 'frame' 'Status Feedback' -----
===== 'program' 'Audiostream' =====
Has been added.
===== 'program' 'Add Task' =====
Has been added.
=====
End comparison at 'Wed Jun 21, 2000 2:13:39 PM EDT'.
2 objects have been added.
0 objects have been deleted.
3 objects have been changed.
11 objects are the same.

```

program1.exc

```

program "Add ECR"
program "Add Note"

```

program3.log

```

Map file 'd:\eMatrix\xml\program1.map' successfully read.
Exclude file 'd:\eMatrix\xml\program1.exc' successfully read.
Input baseline file: 'd:\eMatrix\xml\program1.xml'.
Type = 'program', Name Pattern = 'A*', Workspace 'excluded'.
Start comparison at 'Wed Jun 21, 2000 2:13:28 PM EDT'.
Baseline version was '9.0.0.0'.
Current version is '9.0.0.0'.
=====
===== 'program' 'Add Component (As-Designed)' =====
description has been changed.
was 'Matrix Prof Services: Contains settings for the program to create and connect and
a new object with AutoName logic.'
now 'Matrix Professional Services: Contains settings for the program to create and
connect and a new object with AutoName logic.'
===== 'program' 'Add Assembly (As-Designed)' =====
===== 'program' 'Add Purchase Requisition' =====

```

```

===== 'program' 'Add Event' =====
===== 'program' 'AttributeProg' =====
===== 'program' 'A' =====
===== 'program' 'A1' =====
===== 'program' 'A2' =====
===== 'program' 'Add Assembly' =====
----- 'frame' 'Master Frame' -----
----- 'frame' 'Welcome' -----
----- 'frame' 'Make vs Buy' -----
widget 'MakevsBuy' validateProgram programRef has been changed.
  was 'NameCheckTest'
  now 'NameCheck'
----- 'frame' 'Part Family' -----
----- 'frame' 'Assembly Description' -----
----- 'frame' 'Target Parameters' -----
----- 'frame' 'Status Feedback' -----
'program' 'Add Task' mapped to 'Add Task Import'
===== 'program' 'Add Task Import' =====
----- 'frame' 'Master Frame' -----
----- 'frame' 'Welcome' -----
----- 'frame' 'Key Task Name' -----
----- 'frame' 'Task Description' -----
----- 'frame' 'Status Feedback' -----
===== 'program' 'Add Operation' =====
----- 'frame' 'Master Frame' -----
----- 'frame' 'Welcome' -----
----- 'frame' 'Key Operation Name' -----
----- 'frame' 'Operation Description' -----
----- 'frame' 'Status Feedback' -----
===== 'program' 'Add ECR Import' =====
----- 'frame' 'Master Frame' -----
----- 'frame' 'Welcome' -----
----- 'frame' 'Change Type' -----
----- 'frame' 'Change Class' -----
----- 'frame' 'Reason For Change' -----
----- 'frame' 'Product Line' -----
----- 'frame' 'Change Priority' -----
----- 'frame' 'Reason for Urgency' -----
----- 'frame' 'AdditionalSignatures' -----
----- 'frame' 'Conclusion' -----
----- 'frame' 'Conclusion-Urgent' -----
----- 'frame' 'Status Feedback' -----
===== 'program' 'Audiplay' =====
Has been added.
===== 'program' 'Add ECR' =====
Has been excluded.
===== 'program' 'Add Task' =====
Has been added.
===== 'program' 'Add Note' =====
Has been excluded.
=====
End comparison at 'Wed Jun 21, 2000 2:13:29 PM EDT'.
2 objects have been added.
0 objects have been deleted.
2 objects have been changed.
10 objects are the same.

```

Part III:

Business Administrator Functions

10

User Access

Administrative Objects that Control Access.....	242
Persons.....	242
User Categories.....	243
Policies	244
Rules.....	245
Working With Expression Access Filters	245
Access that is Granted.....	247
Which Access Takes Precedence Over the Other?.....	248
Access Precedence: Description	248
Access Precedence: Flow Chart.....	250
Accesses	251
Summary of All Accesses	251
More About Read Access	255
More About Unlock Access.....	256
More About Show Access.....	256
More about Connection Accesses	258
Summary of User Access Goals	259

Administrative Objects that Control Access

eMatrix lets you control the information users see and the tasks they can perform. Like most Business Administrator functions, you control user access in eMatrix by defining administrative objects in Business Modeler.

This chapter describes all the administrative objects that allow you to control user access. It also defines the various kinds of user access and explains which ones take precedence over others. Use this chapter to help you plan the administrative objects you should create and the accesses you should assign. To implement your plan, refer to the chapter in this guide that describes how to work with each type of administrative object:

To find out how to define:	See:
persons, groups, roles, and associations	Working With Users in Chapter 11
policies	Working With Policies in Chapter 17
rules	Working With Rules in Chapter 16
configurable user interface components	The chapter for the object you want to work with

When a user attempts to perform a task on a business object—for example, view a file checked into the object or change the value of an attribute for the object—the system allows the user to perform the task only if the user has been assigned *access* to perform the task. This section describes the administrative objects in Business Modeler that you can use to control user access. To see a list and descriptions of the accesses you can assign and deny, see [Accesses](#).

If you are working with an application that has a user interface external to eMatrix desktop or Web Navigator, you may also be able to control access by controlling who sees various components of the user interface. For example, if you are building a custom applet, you can use the ADK to determine who sees specific pages and links on the pages. If you are working with an application that is built with configurable component objects, you can use access features of those components to control access. Although configurable component objects are administrative objects that can be used to control access, they are not described here because they are specific to a particular type of user interface. For information, see the *Application Exchange Framework Guide*.

Persons

You must define a person object for every person who will use eMatrix. There are many components to a person definition, such as the user's full name and e-mail address, and several of these components effect user access.

Person Access

Part of creating a person definition involves specifying the accesses the user should have for business objects. If the user will ever need to perform a task for any business object, you must assign the access in the user's person definition. Other administrative objects,

such as policies and rules, allow you to restrict the access that users and user categories (groups, roles, and associations) have for specific business objects and for specific attributes, programs, relationships, and forms.

If you want to prevent a person from ever having a form of access, you may do so by denying that access in the person definition. For example, assume you have a user who continually overrides the signature requirements for a business object. You could prevent the user from ever overriding signatures, even if a policy grants the access to the user, by denying the access in the person definition.

User Type

Another element of a person definition that can effect user access is the user type. If a user's type is System Administrator, eMatrix performs no access checking. Therefore, the system allows such a user to perform any task on any object, even if a policy or rule limits the user's access.

You should assign the System Administrator user type only to people who need full access to all objects, such as a person who will be importing and exporting data or maintaining vaults and stores. For such a user, you should create a second person definition that is not assigned the System Administrator user type. When using eMatrix for routine work with business objects, the user should set the session context using this second person definition.

Also, if the user's type is Trusted, the system allows the user to perform any task that requires read access only (viewing basics, attributes, states, or history for the object).

User Categories

Three administrative objects allow you to identify a set of users (persons) who require the same accesses: groups, roles, and associations. The shared accesses can be to certain types of business objects (as defined in policies) or other administrative objects, such as specific attributes, forms, relationships, or programs (as defined in rules). When you create a group or role, you assign specific users to the category. When you create an association, you assign groups and roles to the association. A user can belong to any number of groups, roles, and associations.

If many users need access to a type of business object, you should consider creating a user category to represent the set of users. Creating a user category saves you the trouble of listing every user in the policy or rule definition. Instead, you just list the user category. For example, suppose you create a user category, such as a group, and assign 25 users to the group. Then you assign the group to a state in the policy and grant the group full access. All 25 users within that group will have full access to objects governed by the policy. It is easier to build and maintain user categories than to specify individual users in all policy and rule definitions.

To decide which kind of user category you should create, consider what the users have in common and why they need some of the same accesses.

- *Groups*—A collection of people who work on a common project, have a common history, or share a set of functional skills.

In a group, people of many different talents and abilities may act in different jobs/roles. For example, an engineering group might include managerial and clerical personnel who are key to its operation. A documentation group might include a

graphic artist and printer/typesetter in addition to writers. While the groups are centered on the functions of engineering or documentation, they include other people who are important for group performance.

- *Roles*—A collection of people who have a common job type: Engineer, Supervisor, Purchasing Agent, Forms Adjuster, and so on.
- *Association*—A collection of groups, roles, or other associations. The members of the user categories have some of the same access requirements based on a combination of the roles users play in the groups in which they belong. For example, perhaps a notification that an object has reached a certain state should be sent to all Managers in an Engineering department. Or maybe, only Technical Writers who are not in Marketing are allowed to approve a certain signature.

Policies

A policy controls many aspects of the objects it governs, including who may access the objects and what tasks they can perform for each state defined in the policy. There are three general categories used to define who may access objects in each state. For each category, you may assign full, limited, or no access.

- *Public*—Everyone in the eMatrix database. When the public has access to perform a task in a particular state, any eMatrix user can perform the task (except if they are denied it in their person definition). When defining public access, it is important to define access limits. Should the public be able to check in files to the object, override restrictions for promotions, and delete objects? These are some of the access questions that you should answer when defining the public access.
- *Owner*—The person, group, role, or association that currently owns the object. When a user initially creates an object, the user (person) who creates it is the *owner*. This user remains the owner unless ownership is transferred to someone else. In an object's lifecycle, the owner usually (though not always) maintains control or involvement. In some cases, the original owner might not be involved after the initial state. Typically, the owner has full access to objects.
- *User*—A person, group, role, or association that has specific access requirements for a particular state. When a group, role, or association is assigned access, all the persons who belong to the group, role, or association will have access. Additionally, all persons assigned to groups and roles that are children of the assigned group or role will have access. (Child groups inherit all accesses from the parent group and child roles inherit all accesses from the parent role.)

For example, you may not want the public to make flight reservations. Therefore, the public is not given access to create reservation objects. Instead, you establish that a Travel Agency group can originate flight reservations. Any member of that group can create a reservation object. Note that once an agent creates a reservation object, that agent is the owner and has all access privileges associated with object ownership.

Assigning user access to groups, roles, and associations is an effective means of providing access privileges to a user. Under most circumstances, a person will have both a group and a role assignment and may also have multiple group and role assignments. In many cases, it is easier to specify the roles, groups, or associations that should have access in a policy rather than list individual users. This way, if personnel changes during a stage of the project, you do not need to edit every policy to change user names.

If a user is assigned access (public, owner, or user) in the current state of an object, the system allows the user to perform the task. For example, suppose a user belongs to a group and a role. If the policy allows the role to perform the task but does not allow the group to perform the task, then the user can perform the task.

Rules

Once eMatrix determines that a user's person definition and the policy allows the user to perform a task for an object's current state, eMatrix then checks to see if any rules prevent the user from performing the task.

As described above, a policy controls the tasks users can perform for each state of an object. In contrast, a rule controls the tasks users can perform regardless of the object. The tasks that rules apply to are limited to only those involving attributes, relationships, forms, and programs. For example, a policy might allow all users in the Engineering group to modify the properties of Design Specification objects when the objects are in the Planning state. But you could create a rule to prevent those users from changing a particular attribute of Design Specifications, such as the Due Date. In such a case, Engineering group users would be unable to modify the Due Date attribute, no matter what type of object the attribute is attached to.

When you create a rule, you define access using the three general categories used for assigning access in policies: public, owner, and user (specific person, group, role, or association). For a description of these categories, see the [Policies](#) section above. Note that owner access does not apply to rules that govern relationships because relationships don't have owners.

Working With Expression Access Filters

User access lists defined on a policy or rule can accept a filter expression in order to grant or deny access to a specific user. If the filter expression evaluates to "true," the specified access will be granted; otherwise the access is denied.

While evaluating expressions that are part of the Expression Access in a policy, Show access is not checked since Show access prevents expands, prints and almost all other commands if the user does not have this access mask. For example, Show access may be one of the accesses that the user would be evaluating for during a traverse through a set of connected business objects.

The following describes operands of expression access filters.

- Anything that one can select on a business object can be used as an operand of an expression access filter. For example:

```
( "attribute[Priority Code]" == "High" ) && (description ~ "*test**")
```

Notice that the business object attribute called "Priority Code" and the business object description are used as operands.

- Anything that one can select from the context user object can be included as an operand of an expression access filter. For example:

```
context.user.isassigned[Group_Name] == true
```

- Any property defined on the "context user" can be used as operand of an expression. For example:

```
context.user.property[Export Allowed].value == true
```

- Dynamic Expressions stored as business object attributes

You can also define an expression access filter in a business object attribute and have eMatrix evaluate the expression stored in the attribute when checking access. For example:

Enter following for the expression filter

```
evaluate attribute[expattr]
```

Evaluate the expression shown below:

```
MQL> evaluate expr 'evaluate attribute[expattr]' on bus ECR1 test 0;
```

TRUE

eMatrix first extracts the expression stored in the attribute “expattr” and then evaluates that expression. Here is what is stored as value of the attribute “expattr”:

```
context.user.isassigned[MX-GROUP-1] == true && context.user.isassigned[Ind_ass] == true
```

- Dynamic Expressions stored as a description of a business object

You can define an expression access filter in a business object description and have eMatrix evaluate the expression stored in the description. For example:

Enter following for the expression filter:

```
evaluate description
```

Evaluate the expression shown below:

```
MQL> evaluate expr 'evaluate description' on bus ECR1 test 0;  
TRUE
```

eMatrix first extracts the description of the business object and then evaluates that expression. Here is what is stored as a description of the business object:

```
context.user.isassigned[MX-GROUP-1] == true && context.user.isassigned[Ind_ass] == true
```

- Dynamic Expressions stored in a description of a connected control object.

Example:

Assembly MTC1234 0 is connected to

Assembly MTC3456 0 by relationship ECR1

Assembly MTC3456 0 is connected to

Part MTC6789 0 by relationship ECR2

Evaluate the expression shown below:

```
MQL> evaluate expr 'evaluate  
to[ECR1].businessObject.to[ECR2].businessObject.description' on bus Assembly "MTC 1234"  
0;
```

eMatrix first extracts the description of the business object “Part MTC6789 0” and then evaluates the expression. Here is what is stored as a description of the business object Part MTC6789 0:

```
context.user.isassigned[MX-GROUP-1] == true && context.user.isassigned[Ind_ass] == true  
This allows storing the expression filter rule in some central to which all objects to be controlled are connected.
```

Expression Access Filter Example

When developing filters, you can use the eval expr command on the filter to qualify that the expression is valid before including it in the policy or rule.

For example:

```
MQL<15> eval expr '(attribute[Actual Weight] > 100) AND ("relationship[Designed Part  
Quantity].to.type" == "Body Shell")' on bus Comment 12345 1;  
FALSE
```

From the above we can say that the expression

```
'(attribute[Actual Weight] > 100) AND ( "relationship[Designed Part Quantity].to.type"  
== "Body Shell")'
```

is a valid expression and thus can be used for an expression access filter.

Access that is Granted

There is one way users can gain access privileges that isn't controlled by the Business Administrator using administrative objects. In eMatrix Navigator, users can grant any or all the access privileges they have for a business object to another user or group. However, a user can only grant accesses on an object to another user or group if the grantor has the "grant" access privilege for the object. As Business Administrator, you control who has grant access by assigning or denying the grant access in the person definition and in policy definitions.

Users can only grant accesses that have been assigned to them in their person definition or in a policy for an object's current state. For example, if a grantor's person definition denies the override access privilege, the user cannot grant that privilege to another user. However, a grantee can be granted privileges that are denied in his/her person definition. (This is the only way users can perform a task that is denied in their person definition.) Such a user could not then grant the privilege to another user.

The MQL command and ADK methods for granting business objects allow users to:

- grant an object to multiple users
- have more than one grantor for an object
- grant to any user (person/group/role/association)
- use a key to revoke access without specific grantor/grantee information

Custom ADK programs and MatrixOne Value Chain Portfolio applications can take advantage of the enhancements. However, desktop eMatrix and eMatrixApplet (Web) user interfaces do not support these changes.

For information on how to grant access to objects, see [Granting Access](#) in Chapter 34.

Which Access Takes Precedence Over the Other?

Suppose a user's person definition does not include delete access but a policy assigns delete access to the user. Will the user be able to delete an object governed by the policy? The answer is no, the user won't be able to perform the action because the person definition takes precedence over the policy. This section describes which kinds of access take precedence over others. To see a flow chart that represents this information, see [Access Precedence: Flow Chart](#).

Access Precedence: Description

The highest level of access control occurs through the user interface: users who have delete access for an object can only delete the object if the user interface provides some mechanism, such as a Delete link, button, or menu option, that lets users delete the object. If you are working with an application, such as a Value Chain Portfolio application, that is built using configurable components (for example, command, menu, table, form objects), you can use access features for these components to control who can see these user interface elements. The access features include restricting user interface components to specific roles or access privileges. Some components also let you control access using a select expression or JPO. For information on the access controls available for configurable components, see "Controlling User Access to User Interface Components" in the *Application Exchange Framework Guide*.

When a user attempts to perform an action for a business object, the system checks to see what type of user is defined in the user's person definition. If the user is a System Administrator-type user, the system allows the action and performs no further checking. If the user is Trusted and the action involves read access only, the action is permitted.

If the user is not System Administrator or Trusted (or if the user is Trusted and the action involves more than read access), eMatrix checks the user's person definition. A user's person definition takes precedence over accesses granted in the policy definition—if an access is denied in the person definition, the user will not have that access, even if a policy assigns the access. However, a person can be “granted” access to a business object even if the action is denied in the person definition. (See [Access that is Granted](#).)

If the person definition allows access, eMatrix next examines the current state of the policy to see if the user is allowed access. The policy allows the user access if the access for the action is assigned to the:

- public *or*
- owner and the user is the owner *or*
- user or to a user category (group, role, association) to which the user belongs

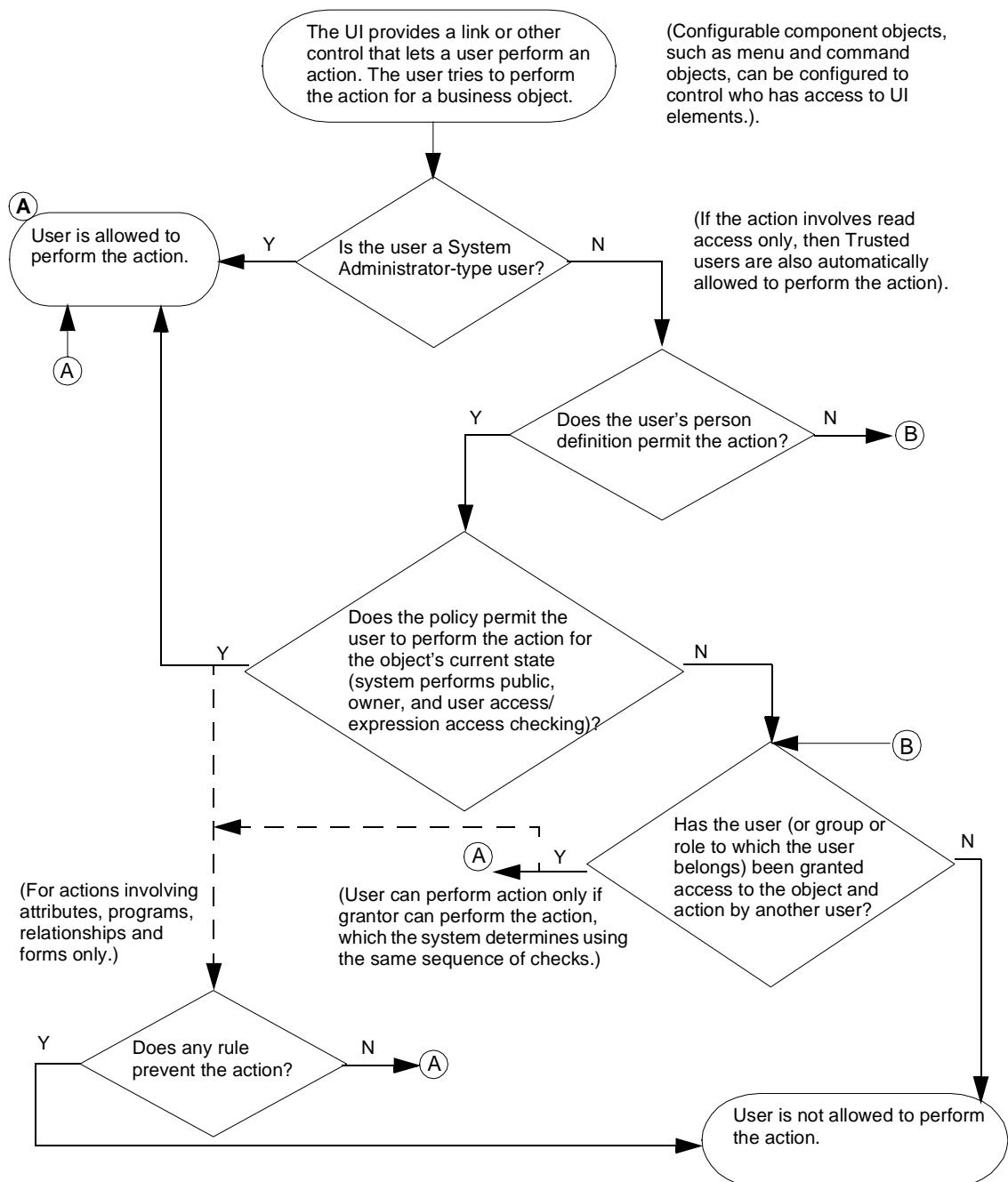
If the user is denied access in the policy or person definition, the system checks to see if the user has been granted the access for the business object by another user. If so, the system makes sure the grantor has the access by going through the same access checking as described above. If the grantor has access, then the user is allowed to perform the action. If the user has not been granted access or the grantor doesn't have the access, the action is denied.

If the policy and person definition allow the access or if the user has been granted access, the user is allowed to perform the action with one important exception. If the action involves an attribute, relationship, form, or program, the system first checks to see if any

rules deny access. If the system finds a rule that governs the attribute, relationship, form, or program for which the action applies, it goes through the same kind of checking as it did for the policy. If the access is assigned for the user in any access (public, owner, or user), the action is allowed. If access is not specifically assigned in a rule that governs the object, access is denied.

WHERE access is found is not as important as *IF* it is found. For example, a user may receive access by virtue of belonging to a group that is assigned to a parent group that is assigned access in a policy. A user's group may be denied access but the user's role is allowed access. In addition, a user might have read access to a business object, allowing attributes or a form to be displayed, but then have modify access to an attribute denied.

Access Precedence: Flow Chart



Accesses

An access is the permission to perform a particular task or action within eMatrix. You assign accesses in person definitions, policies, and rules. In policies and rules, you can assign or deny accesses to the public (all users), owner, or users (persons, groups, roles, and associations). eMatrix users also can grant their accesses for an object to other users. This table lists and describes all the accesses available and shows which administrative object uses each access.

Summary of All Accesses

Access Privilege	Allows a user to:	Can be assigned in:
Read	View the properties of an object, including basics attributes, states, and history. For more information, refer to the More About Read Access , below. To delete files checked into a business object, a user must have read and checkin access.	Person definition Policy definition Rule for attributes
Modify	Edit the attributes of an object or relationship.	Person definition Policy definition Rule for attributes Rule for relationships
Delete	Delete an object from the database. Does not apply to files. To delete files checked into a business object, a user must have both read and checkin access.	Person definition Policy definition
CheckOut	Copy files contained within a business object to the local workstation. Also allows the user to open a file for viewing. To allow a user to use the open for edit command for files checked into an object, the user must have checkin, checkout, and lock access for the object.	Person definition Policy definition
CheckIn	Copy files from the local workstation to a business object. To allow a user to use the open for edit command for files checked into an object, the user must have checkin, checkout, and lock access for the object. To delete files checked into a business object, a user must have both read and checkin access.	Person definition Policy definition

Access Privilege	Allows a user to:	Can be assigned in:
Schedule	Set and modify schedule dates for the states of a business object.	Person definition Policy definition
Lock	<p>Restrict other users from checking files into a business object and from opening files for editing. To allow a user to use the open for edit command for files checked into an object, the user must have checkin, checkout, and lock access for the object. If an object is governed by a policy with enforce locking turned on, users can only lock the object when checking out a file. Users cannot manually lock the object. Enforce locking prevents one user from overwriting changes to a file made by another user. See also the discussion of enforced locking in Enforce Clause in Chapter 17.</p>	Person definition Policy definition
Execute	<p>Execute a program. This access applies only when assigning rules for programs. It does not apply in person or policy definitions. A program rule establishes who has the right to use the programs to which it is assigned by setting owner, public, and user accesses in the rule. Applies only to programs, including wizards, executed explicitly by the user; that is business object methods and those executed via a custom toolbar.</p>	Rule for programs
UnLock	<p>Release a lock placed on a business object <i>by another user</i>. Users may release locks they themselves have placed on objects without this access. Reserve unlock access only for those users who may need to override someone else's lock, such as a manager or supervisor.</p> <p>Unlocking an object locked by another user should especially be avoided for objects governed by policies with enforce locking turned on. For information on enforce locking and how unlocking objects manually can cause confusion, see Enforce Clause in Chapter 17.</p>	Person definition Policy definition
Freeze	<p>Freeze, or lock, a relationship so that business objects may not be disconnected until the relationship is thawed. Also, the type or attributes of a frozen relationship may not be modified.</p>	Person definition Policy definition Rule for relationships

Access Privilege	Allows a user to:	Can be assigned in:
Thaw	Thaw, or unlock, a relationship so that it may be modified or deleted.	Person definition Policy definition Rule for relationships
	<i>Note: When a user attempts to perform a task that requires freeze or thaw access, the system checks the access privileges for the objects on both sides of the relationship (defined in the relevant policies), as well as accesses defined for the relationship type (defined in relevant access rules).</i>	
Create	Create original and clone business objects. Create access applies only for the first state of an object. If a policy gives the owner or the public create access in the first state of an object, anyone will be able to create that type of object (when objects are created, the owner is the one performing the function). To allow only a certain group, role, person, or association to be able to create a specific type of object, deny create access for the owner and public in the object's first state. Then add a user access for the person, role, group, or association that includes create access.	Person definition Policy definition
Revise	Create a revision of a selected business object.	Person definition Policy definition
Promote	Change the state of an object to be that of the next state.	Person definition Policy definition
Demote	Change the state of an object to that of a prior state.	Person definition Policy definition
Grant	Grant the access privileges the user has for a business object to another user.	Person definition Policy definition Rule definition
Revoke	Revoke the access privileges that have been granted for a business object. <i>Use with caution! Anyone with this privilege can revoke grants for another user.</i>	Person definition Policy definition Rule definition

Access Privilege	Allows a user to:	Can be assigned in:
Enable	Unlock the state so that a business object can be promoted or demoted.	Person definition Policy definition
Disable	Lock a state so that a business object cannot be promoted or demoted.	Person definition Policy definition
Override	Disable requirement checking allowing for promotion of an object even when the defined conditions for changing the state have not been met.	Person definition Policy definition
ChangeName	Change the name of a business object.	Person definition Policy definition
ChangeType	Change the type of a business object or relationship. To change a relationship type, the user needs changetype access for the object on both ends of the relationship and on the relationship.	Person definition Policy definition Rule for relationships
ChangeOwner	Change the owner of a business object.	Person definition Policy definition
ChangePolicy	Change the policy of a business object.	Person definition Policy definition
ChangeVault	Change the vault of a business object.	Person definition Policy definition
FromConnect	Link business objects together on the "from" side of a relationship.	Person definition Policy definition Rule for relationships

Access Privilege	Allows a user to:	Can be assigned in:
ToConnect	Link business objects together on the "to" side of a relationship.	Person definition Policy definition Rule for relationships
FromDisconnect	Dissolve a relationship on "from" business objects.	Person definition Policy definition Rule for relationships
ToDisconnect	Dissolve the "to" side relationship between business objects.	Person definition Policy definition Rule for relationships
<i>Note: When a user attempts to perform a task that requires connect or disconnect access, the system checks the access privileges for the objects on both sides of the relationship (defined in the relevant policies), as well as accesses defined for the relationship type (defined in relevant access rules).</i>		
ViewForm	View a form. The ViewForm and ModifyForm accesses apply only when assigning rules for forms. It does not apply to person or policy definitions. A form rule establishes who can view and modify the form to which it is assigned by setting owner, public, and user accesses in the rule.	Rule for forms
ModifyForm	Edit attribute and other field values in a form.	Rule for forms
Show	Control whether a user knows that a business object exists. The access privilege is designed to prevent a user from ever seeing the type, name, or revision of an object.	Person definition Policy definition Rule definition

More About Read Access

The **read** access privilege allows a user to view the properties of an object, including basic information, attributes, states, and history. For example, if a user does not have read access for an object, the user will not be able to see the Object Inspector, Attributes, Basics, States, or History dialog boxes when the object is selected. Furthermore, the user won't be able to expand on the object using the Navigator browser or view the Image assigned to the object. When a user performs queries based on criteria other than type, name, and revision, eMatrix will only find those objects to which the user has read access.

This applies also to visual cue, tip, and filter queries: eMatrix will not apply the visual, filter, or tip to objects found by the query if the user does not have read access to them.

All users have access to tables (both indented and flat) and reports. If a user does not have read access to an object, eMatrix displays only the type, name, and revision. Table cells that provide additional information on objects to which there is no read access display “#DENIED!”. Reports work in a similar manner, providing only the information to which the user has access. MQL commands using `SELECT` or `EXPAND` also return #DENIED! if the user does not have read access. When read access is denied on an attribute, the #DENIED! message is displayed there as well.

If read access to a business object is denied, the system does not display the Attributes or the States browser. This prohibits functions such as modify, promote, demote, and so on. In MQL, however, if users have these privileges, the transactions are allowed. For example, if a user has no read access but does have modify access, the user may modify the object’s attributes in MQL. This is because the user does not have to print the business object (which would not be permitted) before modifying it.

However, if a user has modify or promote access for an object, it makes little sense to deny read access. Similarly, allowing a user to check out files without assigning read access only partially prohibits the user from seeing information for the object. Carefully think through how you assign accesses, whether you are assigning accesses for users (person, group, role, and association), policies, or access rules. Read access must be used logically in conjunction with the other accesses.

More About Unlock Access

Take extra care when granting unlock access. *Without* unlock access, a user can unlock any object locked by that user. Unlock access allows a user to unlock objects that have been locked by *other* users. Reserve unlock access only for those users who may need to override someone else’s lock, such as a manager or supervisor.

Unlocking an object locked by another user should especially be avoided for objects governed by policies with enforce locking turned on. For information on enforce locking and how unlocking objects manually can cause confusion, see [Enforce Clause](#) in Chapter 17.

More About Show Access

When a user does not have show access, eMatrix behaves as if the object does not exist. So, for instance, the `print businessobject` command errors out with the same error as if the object does not exist.

Without show access for an object, the user will never see the object or any information about that object in any browser or in MQL. Specifically, the user will not see the object displayed under these circumstances:

- Performing a “find” in the desktop or Web versions.
- Opening a set.
- Clicking on the plus sign in a Navigator dialog in the desktop or Web clients.
- Evaluating a query in MQL.
- Running the `expand businessobject` command in MQL.
- Running the `print set` command in MQL.
- Running the `expand set` command in MQL.
- Running the `print connection` command in MQL.

- Reading an email message sent via eMatrix that includes a business object that was routed or sent.
- Opening the revisions dialog.
- In the History of an object, other objects are sometimes referred to. The performance impact of determining whether the current user has access to see the Type, Name and Revision of such objects is significant and unavoidable. This performance impact will NOT be resolved by this feature, but individual history records can be deleted using the `delete history` clause of the `modify businessobject` or `modify connection` command. This can be used in action triggers to remove such records.

Additionally, a user who does not have show access for an object will not see references to the object when these operations occur:

- Running the `evaluate expression` command in MQL.
This command evaluates an expression against either a single, named business object or against a collection of them specified via some combination of sets, queries, and/or expansions. In the former case, when a user does not have show access, the command behaves as if the object does not exist. In the latter case, it simply leaves the object out of the collection.
- Running any of the `businessobject` commands in MQL.
- Running various commands against a set, including the `execute`, `modify`, `copy`, and `expand` commands. In general, a “set” command will behave as if the object does not exist in the set.
- If a user has no show access to an object, then the object is not presented as the next or previous revision of an object to which the user does have show access. Such objects are simply left out of (skipped from) the revision sequence. So if there are three objects in a revision sequence with revisions 1, 2, and 3 in that order and the user has show access only to 1 and 3, the next revision of 1 will be represented as 3.
- If the user tries to create an object that exists, but they cannot see, they will receive an error message indicating “Access denied.”

Checking Show Access

The MQL command `set checkshowaccess [ON|OFF]` sets/unsets the global flag to indicate whether the new access checking required by show access is to be executed or not. As this is a one-time-only command, it is provided only via MQL. If set to OFF, the software will not perform the checking, and the feature will not be enabled. If set to ON, the software will always perform the checks.

By default, the database is upgraded with a global setting that indicates that show access should NOT be checked, and therefore the feature will not actually be enabled until this flag is changed. Since show access will not be checked, all users, states and rules will have show access. This will guarantee upward compatibility.

Until the `checkshowaccess` flag is explicitly set to ON, all the new access checks required for this feature will be skipped. Even after the `checkshowaccess` flag is set to ON, the only effect will be the negligible time spent in the new access checks since all users will have Show access to all objects by default.

Show access status is included in export and imports. (However, the access of the global flag is not included).

More about Connection Accesses

Objects can be connected in desktop eMatrix using drag and drop, and the relationship to use may be chosen from the connect bar. You can use the same technique to connect a new object and remove an old object in one step. When replacing an object using drop connect, if you drop onto a child object to replace it, eMatrix keeps the same relationship that was already there rather than using the relationship shown in the connect bar. You are actually *modifying* one end of the connection, not deleting the relationship and creating a new one. This means that to/fromconnect and to/fromdisconnect accesses are checked only on the end of the connection that is changing.

You could create a relationship rule to control this behavior by limiting *modify* access to those who should be allowed to do so. Alternatively, a trigger program could control the ability to perform the replacement.

Summary of User Access Goals

The left column of this table lists goals you may have for controlling user access to information and tasks. The right column summarizes what you need to do to accomplish the goal.

To accomplish this user access goal:	Do this:
Restrict access to a user interface component—such as a menu item, link, table column, or form page row—when the UI is constructed using configurable components	Configure the appropriate dynamic UI object to restrict access based on roles, access privileges, select expressions, or the result of a JPO. For details, see "Controlling User Access to User Interface Components" in the <i>Application Exchange Framework Guide</i> .
Disable read access checking for a single user	In the user's person definition, make the person a Trusted user.
Grant one user's accesses for an object to another user	Have the user grant access to the other user in eMatrix.
Prevent a user from performing a particular task for all objects	Deny access in the user's person definition. For example, suppose you have a user who continually overrides the signature requirements for a business object. You could remove the override access from this person by including !override in the user's person definition. Then, even if the person is allowed this access through a policy, the access is denied.
Control access in different states of an object.	Create groups, roles, and associations that represent a set of users with shared access requirements. Define a policy that allows the public only minimum access and then assign owner and user access as required.
Allow only a certain group, role, person, or association to be able to create a specific type of object	In the policy that governs the object type, edit the access for the first state as follows: <ul style="list-style-type: none">- deny create access for the owner and public- add a user access for the user, role, group, or association and assign create access.
Allow only certain users to execute a program	Define a rule for the program. Assign execute access to the user category that needs to run the program.
Allow all users to view a form but only some to modify it	Define a rule for the form. Assign the viewform access to the public and assign the modifyform access to the user category that needs to edit the form.
Hide an attribute's value from certain users	Define a rule for the attribute. Deny read access to the public and grant it to the user category that should see the value.

To accomplish this user access goal:	Do this:
Hide an attribute from all users	Make the attribute hidden in the attribute definition. When an administrative object is hidden, users don't see any evidence of it in eMatrix.
Allow certain users to create connections of a certain type	Define a rule for the relationship type. Assign create access to the user category that needs to create the relationship, ensuring that they also have toconnect and fromconnect in the policies governing the types at each end.
Allow certain users to remove connections of a certain type	Define an access rule for the relationship type. Assign delete access to the user category that needs to remove the relationship, ensuring that they also have todisconnect and fromdisconnect in the policies governing the types at each end.
Allow certain users to freeze and thaw relationships of a specific type, change the relationship type, and modify attributes	Define an access rule for the relationship type. Assign freeze, thaw, changetype, and modify access to the user category that needs to perform these tasks.

11

Working With Users

Kinds of Users	262
Integrating with LDAP and Third-Party Authentication Tools	263
System-Wide Password Settings	264
Encrypting Passwords	267
Working with Persons	268
Defining a Person	268
Copying and/or Modifying a Person Definition.....	284
Copying (Cloning) a Person Definition.....	284
Modifying a Person Definition	284
Deleting a Person	287
Determining When to Create a Group, Role, or Association.....	288
Working with Groups and Roles	290
Establishing Group and Role Hierarchy.....	290
Defining a Group or Role	291
Modifying a Group or Role Definition.....	295
Deleting a Group or Role Definition	295
Working with Associations.....	298
Uses for Associations	298
Defining an Association	299
Copying (Cloning) an Association Definition.....	302
Modifying an Association Definition	302
Deleting an Association	303
Role-Based Visuals	304
Sharing Visuals.....	304

Kinds of Users

Business Modeler contains four kinds of administrative objects that represent individual users and sets of users: persons, groups, roles, and associations. The primary function of these objects is to allow you to control the information users can see and the tasks they can perform. This chapter explains the circumstances under which you should define these administrative objects and describes how to define each. Before reading this chapter, you should read [Chapter 10, User Access](#) for an overview of how these objects allow you to control user access and how they work with other administrative objects that control access, such as policies and rules.

eMatrixServletCORBA does not support external authentication. Also, external authentication using LDAP integration is not supported for loosely-coupled databases.

Integrating with LDAP and Third-Party Authentication Tools

eMatrix integrates with Lightweight Directory Access Protocol (LDAP) services, so you can use an LDAP service, such as openLDAP or Netscape Directory Server, as a repository to store information about users. The eMatrix integration uses a toolkit from [openldap.org](http://www.openldap.org/) (<http://www.openldap.org/>) for the underlying access protocols and is compliant with LDAPv3. The integration lets you authenticate users based on the users defined in the LDAP database. The integration lets you specify the user information to retrieve from the LDAP service, including address, comment, email, fax, fullname, groups, password, phone, and roles.

eMatrix also lets you authenticate users (persons, groups, or roles) with an external authentication tool such as Siteminder, instead of authenticating through eMatrix. eMatrix provides Single Signon when external authentication is used. This means when a user attempts to access eMatrix (by logging into a Value Chain Portfolio application or the Web version of eMatrix Navigator; external authentication does not apply to the Desktop version) after having been authenticated externally, eMatrix allows the user access and does not present a separate login dialog.

Be aware of the following limitations related to LDAP integration and/or external authentication:

- eMatrixServletCORBA does not support external authentication.
- External authentication using LDAP integration is not supported for loosely-coupled databases.
- LDAP integration is not supported on SGI IRIX or Compaq True 64 operating systems.
- eMatrix CORBA servers do not support the TLS/SSL protocols on HP10.20.
- The integration works with LDAP version 2 servers but version 3 features, such as TLS/SSL, will not work when running on a version 2 server. MatrixOne recommends using version 3 servers.

For details on how to set up integration with an LDAP service and/or an external authentication tool, see the eMatrix Installation Guide.

System-Wide Password Settings

Before defining users, you should consider what your company's password policies are and set system-wide password settings to enforce them. One setting allows you to deny access in the current session to a user who makes repeated failed login attempts. Other settings allow you to control the composition of passwords. For example, you can require that users change their passwords every 90 days, that passwords be at least six characters, and that reusing the old password be prohibited.

The system-wide password settings apply to:

- Every person defined in the database, except users whose person definition includes either the No Password or Disable Password clause
- Every attempt at setting a context, whether the attempt be in eMatrix Navigator, the Business Modeler, the System application, or MQL
- Only passwords that are created or changed after the setting is defined, except for the expiration setting which affects all passwords. For example, suppose you set the minimum password size to 4 characters. From that point on, any password entered in a user's person definition and all new passwords defined by the user in eMatrix Navigator must be at least 4 characters. Any existing passwords that contain less than 4 characters are unaffected. (Tip: You can make passwords for existing users conform to new system-wide password settings by making users change their passwords. Do this for all users by using the `expires` clause, or per user by using the `passwordexpired` clause.)

The following statement allows you to set or change system-wide password settings.

```
set password PASSWORD_ITEM {PASSWORD_ITEM} ;
```

PASSWORD_ITEM is a Set Password clause that provides more information about password settings. You must include at least one clause, and you can include several. The Set Password clauses are:

minsize NUMBER
maxsize NUMBER
lockout NUMBER_OF_TRIES
expires NUMBER_OF_DAYS
[! not]allowusername
[! not]allowreuse
[! not]mixedalphanumeric
[! not]minsize
[! not]maxsize
[! not]lockout
[! not]expires
cipher CIPHER_NAME

Only Business Administrators with Person access are allowed to set system-wide password settings.

Minsize

The Minsize clause of the Set Password statement requires that all passwords be at least a certain number of characters. To remove a minimum size password setting, use the keywords !minsize or notminsize.

Defining a minimum password size of at least 1 ensures that users actually create a password when changing their password. If there is no minimum password size, a user could leave the new password boxes blank when changing passwords, resulting in the user having no password.

Maxsize

The Maxsize clause of the Set Password statement sets an upper limit on the number of characters a password can contain. To remove a maximum size password setting, use the keywords !maxsize or notmaxsize.

For example, to require that users' passwords are least 6 characters and not more than 15, use:

```
set password minsize 6 maxsize 15;
```

By default, passwords are limited to 8 significant characters, in which case a password of 12345678xxxx is the same as password 12345678. The number of significant characters can, however, be controlled using the [Cipher](#) clause of the set password command.

Lockout

The Lockout clause of the Set Password statement prevents a user from logging in after s/he has entered an incorrect password n number of times during a session.

After being locked out, the user's person definition is changed to "inactive." The only way for the user to log in again is to contact the eMatrix Business Administrator to have the setting changed.

In the event that all Business Administrators are locked out, it is possible to resort to the use of SQL to access the database.

To remove a lockout setting, use the keywords !lockout or notlockout.

For example, the following statement allows the user three tries to provide the correct password:

```
set password lockout 3;
```

Expires

The Expires clause of the Set Password statement requires that users create a new password every n number of days. After the specified number of days has elapsed, the system requires users to create a new password in order to log in. To remove the setting, use the keywords !expires or notexpires.

For example, use the following statement if you want users to provide a new password every month:

```
set password expires 30;
```

When you turn on password expiration, passwords that were created prior to version 8 will expire the next time users attempt to log in.

Allowusername

The Allowusername clause of the Set Password statement allows users to create a password that is the same as their username. This is the default. To prevent users from having the same username and password, use the following:

```
set password notallowusername;
```

Allowreuse

The Allowreuse clause of the Set Password statement allows users to enter the same password as their old password. This is the default. To prevent users from keeping the same password, use the following:

```
set password notallowreuse;
```

Mixedalphanumeric

The Mixedalphanumeric clause of the Set Password statement requires that passwords contain at least one number and at least one letter. To remove the setting, use the keyword `!mixedalphanumeric` or `notmixedalphanumeric`.

Cipher

The Cipher clause of the Set Password statement specifies the algorithm used to encrypt passwords.

```
set password cipher CIPHER_NAME;
```

`CIPHER_NAME` is the cipher to be used. It must be one of the LDAP supported ciphers: `crypt`, `md5`, `sha`, `smd5`, `ssha`. The default is `crypt`, which uses only the first eight characters for encryption and comparison.

Setting a new cipher for password encryption does not affect existing passwords. That is, only passwords created or changed after the cipher is specified with the above command will be stored using the new encryption algorithm. To make use of the new cipher, existing users must change their password. Business Administrators can include the [Expires](#) clause when setting the cipher to ensure that all users redefine their password. For example:

```
set password cipher ssha expires 1;
```

After the above statement is issued, existing user passwords will expire in one day, forcing users to enter a new password. Newly defined passwords will be encrypted using the `ssha` cipher.

Encrypting Passwords

Business Administrators can determine which cipher is in use (as well as other system-wide settings) using the MQL `print password` command.

Refer to <http://www.openldap.org/faq/data/cache/346.html> and <ftp://www.ietf.org/rfc/rfc2307.txt> for more information on ciphers.

For LDAP environments, the following MQL command encrypts a password using the same algorithm used for encrypting the eMatrix bootstrap file password. After executing the command, MQL outputs the encrypted text string. Copy and paste it to the file or location where you want to save it.

```
encrypt password PASSWORD_STRING
```

For example, to encrypt the password “secret”, enter:

```
encrypt password secret
```

For details on LDAP authentication, see the *eMatrix Installation Guide*.

Working with Persons

A *person* is someone who uses *any* eMatrix application, not only eMatrix Navigator, Business Modeler, System Manager, and MQL, but also, applications that are part of the eservices application suites available from MatrixOne, as well as those written with the eMatrix ADK. The system uses the persons you define to control access, notification, ownership, licensing, and history.

Two persons are defined when eMatrix is first installed:

- **creator** This person has full privileges for all eMatrix applications.
- **guest** This definition exists for people who use eMatrix infrequently.

You, as the Business Administrator, should first add yourself as a person (Business Administrator) in eMatrix. You should then add a person defined as a System Administrator. (The Business and System Administrators may or may not be the same person.)

You can also define persons who are not users of the system. This is useful for sending notifications to people outside the eMatrix system or for maintaining history records associated with people who no longer work in the organization.

To improve workflow, you may also want to define a person that doesn't represent a real person. For example, you could define a person to represent the company. When objects reach the end of their lifecycle and are no longer actively worked on, you can have the system reassign the objects to this person. Having the company person own inactive objects allows standard users to perform owner-based queries that return only objects that currently require the users' attention. (The Corporate person in eMatrix application suites serves this purpose.)

After you define a person, you can assign the person to user categories (groups, roles, and associations). Use the user categories in policies and rules to control user access.

Defining a Person

There are many parameters that you can enter for a person. While only a name is required, you can use the other parameters to further define the person's relationship to existing user categories, assign accesses, establish security for logging in, as well as provide useful information about the person.

eMatrix users are defined with the Add Person statement:

```
add person NAME [ADD_ITEM {ADD_ITEM}] ;
```

NAME is the name of the person you are creating. All persons must have a unique person name assigned. This name cannot be shared with any other user types (groups, roles, persons, associations). This name will appear in all windows where persons are listed. You could use the system login, if available. In this way you can link the user's context and vault which are associated with the system login. Then, when the person starts an eMatrix session, s/he will automatically begin in his/her context with the specified vault.

The person name is limited to 127 characters. For additional information, see *Business Object Name* in Chapter 34.

ADD_ITEM is an Add Person clause that provides more information about the person you are defining. While none of the clauses is required to make the person usable, they define

the person's relationship to existing groups and roles and provide useful information about the person. The Add Person clauses are:

access ACCESS_MASK { ,ACCESS_MASK}
admin ADMIN_ACCESS_MASK { ,ADMIN_ACCESS_MASK}
address VALUE
assign [group GROUP_NAME] [role ROLE_NAME]
comment VALUE
certificate FILENAME
email VALUE
enable email
disable email
fax VALUE
fullname VALUE
[! not]hidden
icon FILENAME
enable iconmail
disable iconmail
vault VAULT_NAME
site SITE_NAME
password VALUE
no password
disable password
[! not]passwordexpired
phone VALUE
type TYPE_ITEM { , TYPE_ITEM}
property NAME [to ADMINTYPE NAME] [value STRING]

Each clause and the arguments they use are discussed in the sections that follow.

Access Clause

The access clause of the Add Person statement specifies the maximum amount of access a person will be allowed. When this clause is used, you can select from many different forms of access. Each access is used to control some aspect of a business object's use. With each access, other than all and none, you can either *grant* the access or explicitly *deny* the access. You deny an access by entering not (or !) in front of the access mask in the statement.

Business objects are governed by a policy. The policy defines who has access and when. Depending on the business object's state and the person involved, a user may have full, limited, or no access. Generally, the policy that governs the object restricts the access by role or group name. In some cases, that may mean that more access is granted than you might desire for a particular person.

If you want to prevent a person from ever having a form of access, you may do so by denying that access in the person's definition. For example, assume you have a user who continually overrides the signature requirements for business objects. This can be done with an access clause such as:

```
add person George  
access all, notoverride;
```

Even if George is granted override access via a policy, the access will be denied.

For more information on user access, see [Chapter 10, User Access](#). For more information on how access is controlled, see [Working With Policies](#) in Chapter 17.

Access can be assigned or denied using one of two methods:

- **all**—The person has access to *all* functions except those listed.
- **none**—The person has access to *only* the functions listed.

The method you use will depend on whether most access will be permitted or denied. The default is None.

Using All:

```
access all, notchangeowner, notcheckout, notdisconnect, notdelete,  
notdemote, notdisable, notenable, notoverride, notschedule
```

Using None:

```
access none, checkin, connect, create, promote, lock, unlock, modify, read
```

If the amount of access being permitted or denied is about the same, the method does not matter. However, when the amount of access being permitted or denied is heavily weighted toward one or the other, the method you choose can save you time (and typing!).

If the access clause is not included `all` is assumed.

Admin Clause

Users defined as Business and System Administrators can be assigned access to the definitions for which they are responsible. For example, one Business Administrator may be responsible for adding and modifying users; another for the business definitions of attributes, types, policies, etc.; and a third for the programs, wizards, report and forms. All administrators will be able to view all definitions, but create and modify access is controlled by the settings in the Administration Access of the person.

The Admin Clause of the Add Person statement specifies the access that Business and System Administrators will have to business objects. Any of the following accesses can be specified.

Administration Accesses			
Attribute	Association	Site	Menu
Type	Policy	Store	Inquiry
Relationship	Program	Vault	Table
Format	Wizard	Server	
Person	Report	Location	
Group	Form	Process	
Role	Rule		
You can type not or ! at the beginning of an access to explicitly deny the access; for example, !policy or notserver.			

Administrative access can be assigned or denied using all or none in the same manner as when User access is assigned. For example:

```
add person George type business
access all, notoverride admin type attribute
policy;
```

If a person's type is business or system, and the admin clause is not included all is assumed.

Address Clause

The Address clause of the Add Person statement specifies an address for the person. This address could be a mail stop, a street address, or an electronic address. Although this clause is not required, it is helpful to reach the person. An address is limited to 255 characters.

For example, you could assign an address to a user named Wolfgang using either of these statements:

```
add person wolfgang
address "43 Hill Brook Ave, Shelton CT 06484";
Or:
add person wolfgang
address "Mail Stop ES3-A5";
```

Although each example shows a different address, you can include only one Address clause in a person's definition. Remember, though, that the value can be any length. You *could* include all associated addresses in the value of the Address clause. If you do this, remember to place the most important addresses first and to keep it to a minimum to improve readability.

Assign Clause

The Assign clause of the Add Person statement associates the person you are defining with one or more eMatrix roles or groups. (Roles are described in [Working with Groups](#)

and Roles later in this chapter. Although it is not necessary to assign a person to a group or role, it is commonly done to provide easy access to business objects and access privileges.

The Assign clause uses the following syntax:

```
assign [group GROUP_NAME] [role ROLE_NAME]
```

GROUP_NAME is the name of a previously defined group.

ROLE_NAME is the name of a previously defined role.

If either name is not found, an error message is displayed.

When using the Assign clause, you have the option of assigning only a group, a role, or a combination of a role within a group. You can *not* include more than one group or role within a single Assign clause. You can, however, use multiple Assign clauses. For example, each of the following statements include a valid Assign clause:

```
add person myung  
    assign group "Trade Show Support";  
  
add person jenine  
    assign role "Product Demonstrator";  
  
add person jamar  
    assign group "Trade Show Support" role "Trade Show  
Coordinator";
```

Assigning a Role

To assign a role, use the Assign Role variation of the Assign clause. The key to determining whether a person should be assigned to a role is the job s/he performs. The role identifies the person's need to access particular business objects, the amount of access required to do the job, and when the access is needed.

Access to business objects and the files they contain is governed by a policy. The policy can define the groups, roles, and persons that do or do not have access to the business objects (see *Working With Policies* in Chapter 17). The access and amount of access for these classifications can change at various stages of the project life cycle. In many cases, it is easier to specify the roles or groups that should have access in a policy rather than list individual users. If personnel changes during some stage of the project, you do not have to edit every policy to change person names.

To assign a person to a role in the Add Person statement, you use the Assign Role clause:

```
assign role ROLE_NAME
```

ROLE_NAME is the name of a previously defined role. If that role was not previously defined, an error message is displayed.

You can assign a user to a role in two ways, depending on how you are building your database:

- In the person definition, as described here.
- With the Assign Person clause in the Add Role statement described in *Defining a Group or Role*.

If you choose to define the persons first, you can assign them to the role later. If you choose to define persons last, you can make the role assignment in the Add Person statement. Regardless of where you define it, the link between the role and person will appear when you later view either the role definition or the person definition.

Assume you are adding a person named Antonio Pelani with the following statement:

```
add person "antonio pelani"
comment "Directs the allocation and assignment of ER technical staff"
assign role "ER Doctor"
assign role "Lead ER Doctor";
```

In this definition, the defined person is assigned two roles. While the roles are similar, there are distinctions between the Lead ER Doctor and other doctors. Therefore both role assignments are made. Remember that a single person may play many roles in a project. If the role is associated with a policy, it is easier to move a person from one role to another than to edit the policy for each person's name.

Assigning a Group

Persons can be assigned to groups by using the Assign Group variation of the Assign clause. As with assigning roles to a person, assigning a group is not required. However, assigning a group to a person is often the simplest way to assign access privileges to a new user.

In eMatrix, a group identifies a set of users who should share access to selected business objects. In an engineering environment, it might be everyone who is working on a particular project. They would need to have access to drawings and documentation at different stages of the project in order to perform their jobs.

In eMatrix, you can specify which groups have access to business objects and when they have access by using the group name in the policy definition.

As stated in the previous section, access to business objects is governed by a policy. When a group is listed as a valid user in the policy, every person associated with that group has access to all business objects governed by the policy. If you know that a person will work with a group of people in the same function or project, it is easier to assign the person to the group than to edit the policy to add the person's name as a valid user.

To assign a person to a group in the Add Person statement, you use the Assign Group clause:

```
assign group GROUP_NAME
```

GROUP_NAME is the name of a previously defined group. If this name was not previously defined, an error message is displayed.

You can assign a user to a group in two ways, depending on how you are building your database:

- In the person definition, as described here.
- With the Assign Person clause in the Add Group statement described in [Defining a Group or Role](#).

If you choose to define the persons first, you can assign them to the group later. If you choose to define persons last, you can make the group assignment in the Add Person statement. Regardless of where you define it, the link between the group and person will appear when you later view either the group definition or the person definition.

Assume you entered the following Add Person statement:

```
add person sheila
comment "Assesses Training Needs and Implements Classes"
assign role "Training Coordinator"
assign group "Corporate Training"
assign group "Customer Service";
```

In this definition, the person is associated with two groups and one role. The person needs access to the customer service objects in order to work with customers who require training. The person also needs access to the business objects used by the training group. Each of these categories implies different access capabilities.

Since the Training Coordinator role is related to the Corporate Training group, this definition could be simplified by assigning the role with the related group. The following example rewrites the definition using the role and group combination:

```
add person sheila
comment "Accesses Training Needs and Implements Classes"
assign group "Corporate Training" role "Training Coordinator"
assign group "Customer Service";
```

Remember to determine which objects a person will need access to and when that access is required. The answer to those questions can guide you when assigning groups and roles to a person.

Comment Clause

The Comment clause of the Add Person statement provides general information about the person's function and the required privileges. You can have only one Comment clause in any person definition.

There is no limit to the number of characters you can include in the comment. However, keep in mind that the comment is displayed when the mouse pointer stops over the person in the User chooser. Although you are not required to provide a comment, this information is helpful when a choice is requested.

There may be subtle differences in the access privileges required by different users. You can use the Comment clause as a reminder of why this person is defined a certain way. If a person is assigned the wrong group or role, s/he may not be able to fully access the types of business object at the proper times in the object life cycle or may have inappropriate access to an object. Therefore, it is important to completely distinguish all persons.

For example you could have two quality control persons. One person performs testing of component assemblies and the other performs testing of the final machine. While they may belong to the same group (Quality Control) and there are similarities in their roles (Quality Testing), they are unique persons.

To distinguish between persons, you should include any descriptive comments meaningful to you to determine the person to contact when services are required.

For example, in the statements that follow, you can clearly identify the person you would call if you were interested in upgrading your personal computer.

```
add person sandy
comment "Provides PCs sales support";
add person amed
comment "Provides Apple product sales support";
add person miquel
comment "Provides workstations and mainframes sales support";
```

As you can see, each person provides sales support. However, the types of products they serve are different. These clauses enable another eMatrix user to distinguish the persons. The clauses enable you to determine if a person needs access to business objects.

Certificate Clause

Security certificates at the user level are currently not used.

Mail options

When mail is sent, the sender does not know how it is being delivered. The recipient's person definition establishes how it is received—as IconMail, email, both IconMail and email, or neither IconMail nor email.

eMail Clause

The Email clause of the Add Person statement specifies an external electronic mail address. This is an address that is used by the user's non-eMatrix electronic mail utility to connect him/her to other users in the system. Since this address is highly dependent upon the external mail utility, there can be a wide variation in the style and content of the email address. This clause is required if email is enabled (as described below). The email size limit is 127 characters.

The following is an example of an Email clause:

```
add person harriet  
email comments@harriet.com;
```

This email address does not affect the eMatrix internal mail system (IconMail). The eMatrix mail utility uses the person's defined name, role, or group as the address.

Enable Email (Distribution of Mail) Clause

The Enable Email clause of the Add Person statement enables an external electronic mail address for email. The email address specified in the person definition (using the Email clause described above) is used to deliver mail outside of eMatrix. This address is used by the system (not the eMatrix mail utility, IconMail) to connect the user to other users in the system.

You can specify that outgoing messages are sent to email or IconMail (as described for the [Enable Iconmail Clause](#)) or both.

The following is an example of an Enable Email clause:

```
add person harriet  
email comments@harriet.com;  
enable email
```

Since this address is dependent upon the system's mail utility, there can be a wide variation in the style and content of the email address. This address does not affect the eMatrix internal mail system (IconMail). IconMail uses the person's defined name, role, or group as the address.

When a Person is cloned or created with email enabled but no email address has been specified, the following warning is received:

Warning: #1900296: Person 'NAME' has email enabled, but no email address. Use of a fully qualified email address is recommended.

The warning is also shown when a person's email setting is enabled during modification. If an existing Person has email enabled with no address specified, no warning is given if

only other settings are modified. Warnings are also not given when Person's are created during import.

Disable Email Clause

The Disable Email clause of the Add Person statement disables an external electronic mail address for email. The email address specified in the person definition (using the Email clause described above) is not used to deliver mail outside of eMatrix.

Enable Iconmail Clause

The Enable Iconmail clause of the Add Person statement specifies that outgoing messages are sent to IconMail. You can send messages by using email, IconMail or both.

The following is an example of an Enable Iconmail clause:

```
add person harriet  
email comments@harriet.com  
enable iconmail;
```

eMatrix IconMail uses the person's defined name, role, or group as the address.

Disable Iconmail Clause

The Disable Iconmail clause of the Add Person statement disables IconMail so that messages are not received via IconMail.

Fax Clause

The Fax clause of the Add Person statement associates a fax number with the person. You can include additional information with the fax number. The Fax clause is limited to 64 characters. For example, the following are valid uses of the Fax clause:

```
add person shandrika  
fax "(203) 987-5584";  
  
add person "A-1 Consulting Agency"  
fax "(617) 535-9857 please call before faxing";
```

Fullname Clause

The Fullname clause of the Add Person statement specifies the full name of the person you are defining. This name could be the name of the person as it appears in personnel records or the person's signature name. Often when defining a person, the name is abbreviated or shortened. You can include the person's full name as well with the Fullname clause. The full name is limited to 127 characters.

All persons must have a name assigned. When you create a person, you should assign a name that has meaning to both you and the user. If the number of users is small, you may want to use only the first or last name as the person name. If there are larger numbers of eMatrix users, you may want to use the full name or a name and initial to help distinguish users. For example, each of the following is valid person name:

```
Pat M.  
Patty  
Patricia L. Melrose
```

```
P. Melrose  
Pat the Manager
```

Hidden Clause

You can specify that the new person object is “hidden” so that it does not appear in the User chooser in eMatrix. Users who are aware of the hidden person’s existence can enter its name manually where appropriate. Hidden objects are accessible through MQL. For example:

```
add person patty hidden;
```

Icon Clause

Icons help users locate and recognize items by associating a special image with a person, such as a picture of the individual you are defining. You can assign a special icon to the new person or use the default icon. The default icon is used when in view-by-icon mode. Any special icon you assign is used when in view-by-image mode. When assigning a unique icon, you must use a GIF image file. Refer to [Icon Clause](#) in Chapter 1 for a complete description of the Icon clause.

GIF filenames should not include the @ sign, as that is used internally by eMatrix.

Password Options

The use of passwords is similar to the way that passwords work on most systems. Passwords are an effective means of preventing unauthorized access to business objects. Without passwords, any eMatrix user could set their context to that of any other eMatrix user. This would essentially make policies and states meaningless since you could easily bypass access restrictions by pretending to be someone else. Therefore, a password should be required in order to ensure that the person who is logging in is indeed that person. In particular you should thoroughly consider the option of passwords for Business and Systems Administrators.

There are three clauses that use passwords to restrict access to a person’s context:

- Password clause assigns an eMatrix password to the person you are defining (as described below).
- Disable Password clause restricts access to the user whose system ID or login matches the person ID (as described in [Disable Password Clause](#)).
- No Password clause specifies that no password is required to set context. (as described in [No Password Clause](#)).

Only one password-related clause (Password, Disable Password, or No Password) should be given.

```
add person dave  
no password  
access checkin, create, delete, read, modify, checkout,  
connect;
```

Each user can redefine his/her own password by using the eMatrix Preferences (password) option in eMatrix Navigator.

Password Clause

The Password clause assigns an eMatrix password to the person you are defining. The password will be required whenever someone wants to access this person's business objects. When a password is assigned, anyone who knows the password can set their context to this person.

For example, the following statement defines a person with a password value:

```
add person chris  
    password SaturnV;
```

If anyone wanted to set her/his context to Chris, s/he would need to know the password SaturnV.

The defined password is not visible to anyone. As a Business Administrator, you can change a password without seeing or knowing it. The password can contain up to 127 characters, including spaces.

By default, passwords are limited to 8 significant characters, in which case a password of 12345678xxxx is the same as password 12345678. The number of significant characters can, however, be controlled using the [Cipher](#) clause of the `set password` command.

Disable Password Clause

The Disable Password clause lets you use the security for logging into the operating system as the security for setting context in eMatrix. When a user attempts to set context in eMatrix, the system compares the user name used to log into the operating system with the list of persons defined in eMatrix. If there is a match, and the password is disabled, the system sets context and the set context window is not displayed. If there is not a match, or any other password option is used for the matching person, the set context window is presented.

When Disable Password is chosen for an existing person, eMatrix modifies the password so that others cannot access the account. This is similar to the way automatic SSO-based user creation is handled. To re-enable a password for such a person, create a new password for the person as you normally would. See [Setting Context With Disabled Passwords](#) in Chapter 31 for examples.

No Password Clause

The No Password clause specifies that no password is required to access the person's business objects. When No Password is specified, other eMatrix users can set their context to that of the person you are defining. Once the context is set to the person, the eMatrix user can act as that person with all applicable access privileges.

The No Password clause generally is not recommended since removing the password requirement permits other eMatrix users to set context as that user. However, if a person leaves the company, for example, or if you are setting up a guest account, you may want to remove the password requirement.

Passwordexpired Clause

When the Passwordexpired clause is included in a person's definition, the system requires the person to define a new password the next time s/he attempts to log in. The system will

not allow the person to log in without entering a new password. After the user defines a new password, the clause is removed from the person definition.

For example, you could use the following statement to ensure that Jordan defines a new password on next login:

```
add person jordan passwordexpired;
```

This option allows you to require users to establish a password without requiring you to assign them one now. Having users establish their own passwords helps them remember their password. It also prevents you from having to enter an initial password for every new user, which you would then have to communicate to users. You can also use this clause if you suspect that a person's password has been compromised.

Phone Clause

The Phone clause of the Add Person statement associates a telephone number with the person you are defining. This number could be an internal extension number, business phone number, car phone number, or home phone number. The Phone clause is restricted to 64 characters.

For example, you could assign a phone number as follows:

```
add person andrea
    phone "business: x433, home: (203) 987-6543";
```

Notice that you can include additional information about the number to guide those who might reference it.

Property Clause

Integrators can assign ad hoc attributes, called Properties, to the person. Properties allow associations to exist between administrative definitions that aren't already associated. The property information can include a name, an arbitrary string value, and a reference to another administration object. The property name is always required. The value string and object reference are both optional. The property name can be reused for different object references, that is, the name joined with the object reference must be unique for any object that has properties.

```
add person NAME
    property NAME [to ADMINTYPE NAME] [value STRING];
```

For additional information on properties, see *Overview of Administration Properties* in Chapter 24.

Using INI Settings as Properties

Many of the settings in the matrix.ini file are intended for use in a desktop client environment to allow a user to set personal preferences. However, in a client-server environment, the single ematrix.ini file read by the Collaboration Server cannot fulfill this role for all users connected as clients. So, you can use properties on person objects to provide values for .ini settings that are intended to express a personal preference.

Only a few .ini variables are appropriate to set as person properties for the Web version of eMatrix Navigator and Value Chain Portfolio application users. They are:

MX_DECIMAL_SYMBOL
MX_SITE_PREFERENCE

MX_ALIASNAME_PREFERENCE (see [Enabling Aliases](#) in Chapter 25)
MX_LANGUAGE_PREFERENCE (see [Enabling Aliases](#) in Chapter 25)

Site Clause

A site is a set of locations. It is used in a distributed environment to indicate the file store locations that are closest to the group. The Site clause specifies a default site for the person you are defining. Consult your System Administrator for more information.

To write a Site clause, you will need the name of an existing site. If you are unsure of the site name or want to see a listing of the available sites, use the MQL List Site statement. This statement produces a list of available sites from which you can select. (Refer to [Defining Sites](#) in Chapter 6.)

Sites may be set on persons, groups and roles, as well as on the Collaboration server (with MX_SITE_PREFERENCE). The system looks for and uses the settings in the following order:

- if using a Collaboration Server, the MX_SITE_PREFERENCE is used.
- if not using a Collaboration Server, or the MX_SITE_PREFERENCE is not set on the server, if there is a site associated with the person directly, it is used.
- if no site is found on the person, it looks at all groups to which the user belongs. If any of those groups have a site associated with it, the first one found is used.
- if no sites are found on the person or their groups, it looks at all roles the person is assigned. If any of those roles have a site associated with it, the first one found is used.

Add the MX_SITE_PREFERENCE variable to the Collaboration Server's initialization file (bos.ini or ematrix.ini). This adjustment overrides the setting in the person, group, or role definition for the site preference, and should be set to the site that is local to the server. This ensures optimum performance of file checkin and checkout for Web clients.

Type Clause

The Type clause specifies the type of eMatrix user the person you are defining will be. There are four basic types of users. Each type has a level of access to eMatrix:

Application User	A licensed user that can access the eMatrix Database only through the eMatrix Collaboration Server or MQL. All other active users have Application User privileges automatically.
Full User	A licensed user with normal user access.

Business Administrator	A licensed user with access to the Business Administrator functions. When a person is defined as a Business Administrator, s/he may have the privilege of being able to set context to any defined person without a password depending on the system setting for privilegedbusinessadmin. Refer to Privileged Business Administrators in Chapter 3 for more information.
System Administrator	A licensed user with access to the System Administrator functions.
Inactive	A defined user who does not currently have access to eMatrix.
Trusted	A licensed user for whom read access is not checked.

Business and System Administrators are also assigned as Full Users.

All persons have a type. If the type is not explicitly assigned, the person will be automatically assigned to the types application and full. To assign a type, write a Type clause using the following syntax:

type TYPE_ITEM { , TYPE_ITEM}

TYPE_ITEM assigns or denies the privileges associated with each of the five user types. There are twelve TYPE_ITEMS. Six of the items assign the user type and six remove a user type assignment:

TYPE_ITEMS	The user has...
application	Access only through the Collaboration Server or MQL
notapplication	No access
full	Normal access
notfull	No normal access
business	Access to the Business Administrator functions
notbusiness	No access to Business Administrator functions
system	Access to the Business and System Administrator functions
notsystem	No access to the System Administrator functions
notinactive	Current access
inactive	No current access
trusted	Read access that is not checked
nottrusted	Read access that is checked

When you define a person, eMatrix automatically assigns the person a type of Application User and Full User. This means that the user is defined as:

type full, application

If you want a type that is different from this, you need to write a Type clause. For example, the following statement defines a person who is both a Full User and a Business Administrator:

```
add person rodolph  
type full, business;
```

If you want the person's type to be something other than Full and Application, be sure to use the 'not' prefix on any types defined by default. For example, to define a consultant named Feng that is a full user but not and application user, use the following:

```
add person Feng  
type notapplication;
```

Of all the person types, the Inactive type seems unnecessary. If a person is not allowed to access eMatrix, why have that person defined within the database? One answer has to do with business object creation and ownership. Let's assume that you have an employee who worked for some time within the eMatrix system. After that person left the company, you still have many business objects that were created and controlled by the person.

Rather than change object ownership, you may want to maintain the old ownership so that you have a record of the original creator. By making the person *inactive*, you remove that person's ability to access eMatrix while maintaining the status of all business objects the person created. Although access for other users remains as defined, if an owner is inactive the Business Administrator may want to reassign ownership to another person.

Vault Clause

The vault clause specifies a default vault for the person you are defining. It is similar to setting a default directory in your operating system. When a person invokes eMatrix, the context dialog fills in this vault as the default. The vault should be the one most commonly used by the person. Although the person may change to a different vault once eMatrix is started, it is helpful to set the vault that the person should start within.

To write a Vault clause, you will need the name of an existing vault. If you are unsure of the vault name or want to see a listing of the available vaults, use the MQL List Vault statement. This statement produces a list of available vaults from which you can select.

To reduce network loads, vaults are often created locally, serving selected groups of eMatrix users. When you are assigning the vault, you should determine if there is a vault local to the person you are defining. If there is more than one local vault, determine which is better suited for the person based on the type of work or objects used.

For example, the following person definition assigns the engineer to a vault related to the types of projects she will work on:

```
add person mcgivern  
fullname "Jenna C. McGivern"  
assign role Engineer  
assign group "Building Construction"  
vault "High Rise Apartments";
```

In an LCD environment in particular, a default vault setting for every user is recommended. Each federation has its own administrative definitions, which are created in its own ADMINISTRATION vault. An object's vault is used to determine its "home" database, and therefore it is necessary to establish a legitimate business object vault before selecting the type chooser or policy chooser buttons when creating a business object (original, revision or clone) in order for eMatrix to show the correct types or policies.

If a user does not have a default vault, and sets context without a vault, the ADMINISTRATION vault is displayed in the vault field in the create window. If the user then attempts to bring up the type or policy choosers, they will receive the following error:

Vault name is invalid.

This situation can be avoided by giving each user a default vault.

Copying and/or Modifying a Person Definition

Copying (Cloning) a Person Definition

After a person is defined, you can clone the definition with the Copy Person statement. This statement lets you duplicate person definitions with the option to change the value of clause arguments:

```
copy person SRC_NAME DST_NAME [MOD_ITEM {MOD_ITEM}];
```

SRC_NAME is the name of the person definition (source) to be copied.

DST_NAME is the name of the new definition (destination).

MOD_ITEMS are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

Modifying a Person Definition

After a person is defined, you can change the definition with the Modify Person statement. This statement lets you add or remove defining clauses and change the value of clause arguments:

```
modify person NAME [MOD_ITEM {MOD_ITEM}];
```

NAME is the name of the person to modify.

MOD_ITEM is the type of modification to make. Each is specified in a Modify Person clause, as listed in the following table. Note that you need to specify only fields to be modified.

Modify Person Clause	Specifies that...
access ACCESS { ,ACCESS}	The person is restricted to the listed access. Values for ACCESS can be found in the table in <i>Accesses</i> in Chapter 10.
admin ADMIN_ACCESS { ,ADMIN_ACCESS}	The Business or System Administrator is restricted to the listed access. Values for ADMIN_ACCESS can be found in the <i>Admin Clause</i> section.
address VALUE	The address to associate with the person is changed to the value entered. This address could be the person's street address.
assign [group GROUP_NAME] [role ROLE_NAME]	The person is associated with the listed group or role.
comment VALUE	The current comment, if any, is changed to the value entered.
certificate FILENAME	The current certificate is changed to the file specified.
add certificate FILENAME	The named certificate file is added to the person definition.
remove certificate FILENAME	The named certificate file is removed from the person definition.

Modify Person Clause	Specifies that...
revoke certificate FILENAME	The current certificate is revoked.
email VALUE	A valid electronic mail address is set for the person. This address must be in a form understood by the user's e-mail utility.
enable email	Incoming messages are sent to the e-mail address specified with the Email clause.
disable email	Incoming messages are not sent to the e-mail address.
fax VALUE	The person's fax number is changed or is set to the value entered.
fullname VALUE	The full name of the person is changed or is set to the value entered.
icon FILENAME	The image is changed to the new image in the file specified.
enable iconmail	Incoming messages are sent to IconMail.
disable iconmail	Incoming messages are not sent to IconMail.
vault vault_NAME	The current vault is changed to the new vault.
name VALUE	The current person name is changed to the new name
password VALUE	When setting context, the password is changed to the value entered.
no password	When setting context, there is no password required to access this person's context.
disable password	Access may be set to the person only from a machine where the O/S user is the same as the eMatrix user.
passwordexpired	When setting context, a person must define a new password.
phone VALUE	The person's phone number is changed or is set to the value entered.
assign [group GROUP_NAME] [role ROLE_NAME]	The person is assigned to the specified group or role.
assign all	The person is assigned to all groups and roles.
remove [group GROUP_NAME] [role ROLE_NAME]	The person is removed from the specified group or role.
remove all	The person is removed from all groups and roles. Any links the person has with any groups or roles is dissolved.
type TYPE_ITEM { , TYPE_ITEM}	The person is assigned or denied the privileges associated with the listed user types. Values for TYPE_ITEM can be found in the <i>Type Clause</i> section.
hidden	The hidden option is changed to specify that the object is hidden.

Modify Person Clause	Specifies that...
nohidden	The hidden option is changed to specify that the object is not hidden.
property NAME [to ADMINTYPE NAME] [value STRING]	The named property is modified.
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.

As you can see, each modification clause is related to the clauses and arguments that define the person. For example, assume you want to alter the address and phone number of a person who moved. You could do so by writing a Modify Person statement similar to the following:

```
modify person clinton
    address "White House, Pennsylvania Ave, Washington"
    phone "Unlisted";
```

This statement changes the address to that of the White House and designates the current phone number as “Unlisted.”

When modifying a person:

- The roles or groups that you assign to the person must already be defined within the eMatrix database. If they are not, an error will display when you try to assign them.
- Note how access privileges are shared between roles and groups. Although a person is restricted access because of the group assignment, the user may have access via his/her role assignment. To restrict a user that has access or increase a person’s access, you will have to determine the best method for giving that access.

Remember that altering the group or role access affects everyone associated with that group or role. If it is a singular case of special access, you may want to assign that person to the policy directly or define a role that is exclusively used by the person in question.

Deleting a Person

If a person leaves the company or changes jobs so that s/he no longer needs to use the eMatrix database, you can delete that person by using the Delete Person statement:

```
delete person NAME;
```

NAME is the name of the person to be deleted.

When this statement is processed, eMatrix searches the list of defined persons. If the name is not found, an error message is displayed. If the name is found, the person is deleted only if there are no business objects that are owned by the person. If the name is still attached to any objects, the person cannot be deleted.

If the person has created an extensive number of objects or has been heavily involved in the history of many objects (such as a manager signing off), you may want to make the person inactive rather than delete him/her. By assigning an Inactive type to the person, you have a point of reference when that user name comes up in history records or elsewhere.

In addition, IconMail references to the deleted person cause the mail to be unreadable. This includes messages sent by the person, or cc'd to the person. If the person used IconMail extensively, it is better to make the person Inactive than to delete it.

When a person is deleted, any linkages to that person are dissolved. That means the person is automatically removed from any role or group that was included in the person's definition. The person is removed from any signatures in all policies, and if s/he was the only user referenced in the signature, it is removed as a requirement.

For example, if you wanted to delete the person named jones, you would enter the following MQL statement:

```
delete person jones;
```

After this statement is processed, the person is deleted and you receive an MQL prompt for another statement.

Determining When to Create a Group, Role, or Association

If many users need access to a type of business object, you should consider creating a group, role, or association to represent the set of users. Creating a user category saves you the trouble of listing every user in the policy or rule definition. Instead, you just list the user category.

To decide which kind of user category you should create, consider what the users have in common and why they need some of the same accesses.

- *Groups*—A collection of people who work on a common project, have a common history, or share a set of functional skills.

In a group, people of many different talents and abilities may act in different jobs/roles. For example, an engineering group might include managerial and clerical personnel who are key to its operation. A documentation group might include a graphic artist and printer/typesetter in addition to writers. While the groups are centered on the functions of engineering or documentation, they include other people who are important for group performance. They all need access to common information.

- *Roles*—A collection of people who have a common job type: Engineer, Supervisor, Purchasing Agent, Forms Adjuster, and so on.
- *Association*—A combination of groups and roles, or other associations. The members of an association have some of the same access requirements based on a combination of the roles users play in the groups in which they belong. For example, perhaps a notification that an object has reached a certain state should be sent to all Managers in an Engineering department. Or maybe, only Technical Writers who are not in Marketing are allowed to approve a certain signature.

Groups, roles, and associations can be used in many ways to identify a set of users.

- As recipients of IconMail sent manually or sent automatically via notification or routing of an object as defined in its policy.
- As the designated Approver, Rejecter, or Ignorer of signatures in the lifecycle of an object.
- In the user access definitions in the states of a policy.
- As object owners, through reassign.

There are just a few differences between the three categories:

- Users can grant their accesses to objects to persons and groups, but not to roles and associations.
- Persons can be assigned directly to groups and roles. Groups and roles make up associations. So a person belongs to an association by virtue of belonging to a group or role that is assigned to the association.
- Groups and roles can be hierarchical. A group or role can have multiple parents and multiple child groups/roles. Associations are not hierarchical.

The key to determining who should be in which group depends on:

- Whether the user requires access to the business objects.
- How much access is required for the user to do his/her job.

- When access is needed.

For example, suppose a company is manufacturing a new product, called product ABC. Depending on where the project is in its lifecycle, different people will work on the project, require access for different tasks, or use information related to it:

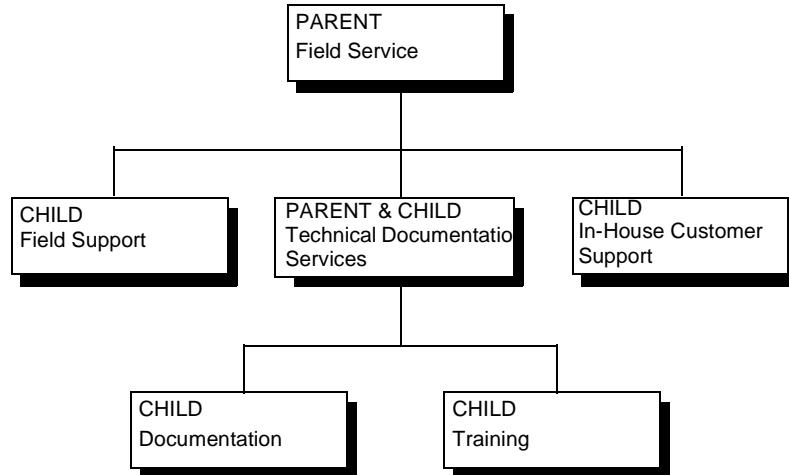
Lifecycle	Users Needing Access
Planning	<p><i>ABC Product Manager association</i>—An association made up of persons who belong to the Product Manager role and the Product ABC group. The Product Managers create the schedule and project plan and write specifications.</p> <p><i>Product ABC group</i>—All persons involved with the product. This group will view the schedules and plans prepared by the Product Managers.</p>
Review	<p><i>Quality Assurance role</i>—Persons who test products for the company. These people view the specifications and write test plans.</p> <p><i>Product ABC group</i>—Review the specifications prepared by the Product Managers.</p> <p><i>ABC Product Manager association</i>—Review and update items as needed.</p>
Release	<p><i>Implementation group</i>—Engineers, clerical help, a financial manager, and a supervisor help implement the product after it is sold. This group includes some individuals from the user categories assigned to the first two states. For these individuals, you can change the amount of access they have to reflect their changing tasks.</p> <p><i>ABC Marketing association</i>—Persons who belong to both the Marketing Manager role and to the Product ABC group. Advertise the product and manage order processing.</p>

Working with Groups and Roles

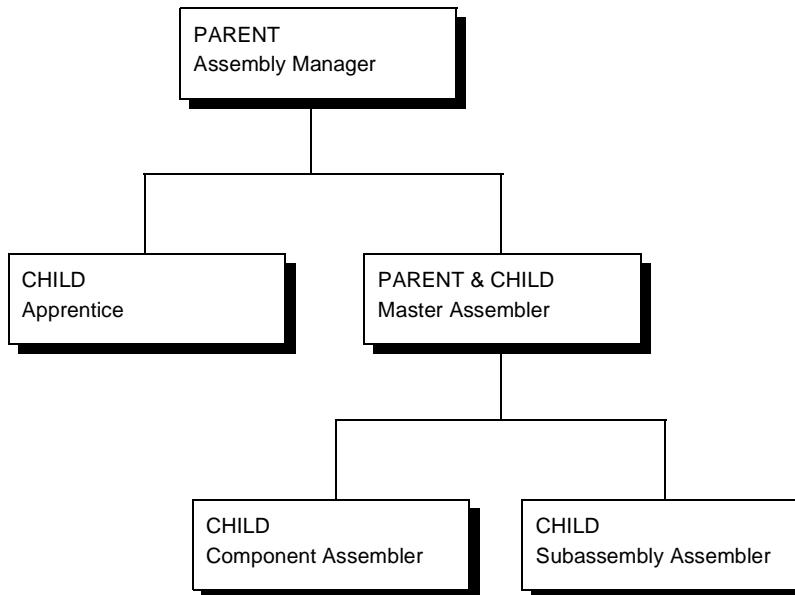
Establishing Group and Role Hierarchy

Groups are frequently hierarchical. In hierarchical groups, access privileges that are available to a higher group (*parent group*) are available to all of the groups below it (subgroups or *child groups*).

For example, in the figure below, the Field Service group is the parent of the Field Support, Technical Documentation Services, and In-House Customer Support groups. The Technical Documentation Services group is also a parent of the Documentation and Training groups.



Roles are often similarly hierarchical. For example, in the figure below, the Assembly Manager role is the parent of the Apprentice and Master Assembler roles. The Master Assembler role is also a parent of the Component Assembler and Subassembly Assembler roles.



A child group or role can have more than one parent. Child groups/roles with more than one parent inherit privileges from each of the parents. You can establish group and role hierarchy using the procedures in this chapter.

Defining a Group or Role

The clauses for defining groups and roles are almost identical. While only a name is required, the other parameters can further define the relationships to existing users, as well as provide useful information about the group or role.

Groups and Roles are created and defined with one of following MQL commands:

add group NAME [ADD_ITEM {ADD_ITEM}] ;
add role NAME [ADD_ITEM {ADD_ITEM}] ;

ADD_ITEM is a clause that provides more information about the group or Role you are creating. While none of the clauses are required to create the group or role, they are used to assign specific users and roles to the group or users and groups to the role. The ADD_ITEM clauses are:

description VALUE
icon FILENAME
parent GROUP_NAME{ ,GROUP_NAME} (For Group)
parent ROLE_NAME{ ,ROLE_NAME} (For Role)
child GROUP_NAME{ ,GROUP_NAME} (for Group)
child ROLE_NAME{ ,ROLE_NAME} (for Role)
assign person PERSON_NAME [role ROLE_NAME] (for Group)
assign person PERSON_NAME [group GROUP_NAME] (for Role)
site SITE_NAME
[! not]hidden
property NAME [to ADMINTYPE NAME] [value STRING]

Each clause and the arguments they use are discussed in the sections that follow.

Defining the Name

You specify the name of the group or role you are creating with the add command. The name will appear whenever the group is listed in a browser. This name must be unique and cannot be shared with any other type of user (groups, roles, persons, associations). Assign a name that has meaning to both you and users. The name field limit is 127 characters. For additional information, refer to [Business Object Name](#) in Chapter 34.

For example:

add group "Insurance Sales Team";
add role "Graphic Artist";

Icon clause

You can assign a special icon to the new group or role. Icons help users locate and recognize items. When assigning an icon, you must select a special GIF format file, as described in [Icon Clause](#) in Chapter 1.

The icon assigned to a group or role is also considered the ImageIcon of the object. When an object is viewed as either an icon or ImageIcon, the GIF file associated with it is displayed.

Description Clause

A description provides general information about the function of the group or role and applicable privileges.

There is no limit to the number of characters you can include in the description field. However, keep in mind that the description appears when the mouse pointer stops over the group or role in the User Chooser. Although you are not required to provide a description, this information is helpful when a choice is requested.

For example, you could have two Technical Writing groups. One group edits user manuals and the other produces the finished book. Some people may work in both groups and there may be similarities in their skills. However, they are unique groups. For example:

```
add group "Technical Writing Editors" description "Editors who mark-up user manuals.";  
add group "Desktop Publishing" description "Desktop publishing editors who implement final edits for production.";
```

Additionally, you could have two types of engineering roles: Hardware and Software. While they are both engineers and will use some of the same business objects in their work, there are differences in the type of work they do. Even among Hardware Engineers, you may have levels based on experience and specific skills. What is the difference between the Associate Engineer and the Principal Engineer? The answer should be reflected in the role descriptions. For example:

```
add role "Associate Software Engineer" description "Has limited direct experience & technical background in the dev. of software applications.";  
add role "Principal Software Engineer" description "Has direct experience & technical background in developing software applications.";
```

Parent and Child Clauses

The Parent and Child clauses of the add group and add role commands define the relationship of the new group or role to other defined groups. This hierarchy allows one group or role to share access privileges with another, saving time when defining access privileges when a policy is defined. You can have any number of child groups and any number of parent groups. For example:

```
add group "Technical Marketing" parent Marketing;  
add role "Engineer" child "Principal Engineer";  
add group "Quality Engineering Managers" parent Engineering,Management;  
add role "Engineering Manager" parent Manager,Employee;
```

Of course, the group or role named as the parent or child must be previously defined.

Refer to [Establishing Group and Role Hierarchy](#) for more information.

Assign clause

The Assign Person clause assigns specific users to the group or role. A group or role can have no users, or they could have many. Groups or Roles with no users may be defined to show a hierarchical relationship between groups or roles. In that case, the defined group or role acts as a parent for other groups or roles.

Most groups and roles will have users assigned to them. When assigning users, the number is limited only by the maximum number defined in the database. Use the Assign Person clause to assign a person to a group or role:

assign person PERSON_NAME [role ROLE_NAME]
assign person PERSON_NAME [group GROUP_NAME]

PERSON_NAME is the previously defined name of a person.

ROLE_NAME is the previously defined role.

GROUP_NAME is the previously defined role.

If these names are not previously defined, an error message is displayed.

You can assign users groups and roles in two ways, depending on how you are building your database:

- With the Assign Person clause in the add Group or add Role command, as described here.
- In the Person statement described in [Working with Persons](#).

Since previously defined names are required to make the Assign Person clause valid, it is not uncommon to wait before assigning persons to a role or group definition. When building the database, you may want to define only the roles and groups and then handle the assignment of users in the person definitions. But, if you choose to define the persons first, or if you are adding a role or group to an existing database, you can assign users to the group or role with the assign person clause. Regardless of where you define it, an assignment made to a role or group becomes visible from all applicable definitions. This means that the link between the group or role and the person will appear when you later view either the group or role definition or the person definition.

When assigning a person to a group, you can also define that person's role within the group. Likewise when assigning a person to a role, you can also define that person's group. There is no special distinction made between a person assigned a role and a person assigned a role within a group. Including a role assignment with the person assignment serves as another means of making a role assignment to a person. Again, once the assignment is made, it can be seen in all person, group, and role definitions when the definitions are viewed. Use the method and statements most convenient for your application and database.

For example, assume you want to add a role called “Trade Show Support” that associates members of the sales and customer service groups with it. You could write a command similar to the following:

```
add role "Trade Show Support"
description "Personnel for Trade Show Support"
child "Trade Show Backup Support"
parent Marketing
assign person elsie group "Sales Force"
assign person mark
assign person richard
assign person jenine;
```

When this statement is processed, the role of “Trade Show Support” is assigned to four persons. After this statement is executed, you can examine the person definitions to see a role assignment included in the definition.

Site Clause

A site is a set of locations. It is used in a distributed environment to indicate the file store locations that are closest to the group. The Site clause specifies a default site for the group or role you are defining. Consult your System Administrator for more information.

To write a Site clause, you will need the name of an existing site. If you are unsure of the site name or want to see a listing of the available sites, use the MQL List Site statement. This statement produces a list of available sites from which you can select. (Refer to [Defining Sites](#) in Chapter 6.)

Sites may be set on persons, groups and roles, as well as on the Collaboration server (with MX_SITE_PREFERENCE). The system looks for and uses the settings in the following order:

- if using a Collaboration Server, the MX_SITE_PREFERENCE is used.
- if not using a Collaboration Server, or the MX_SITE_PREFERENCE is not set on the server, if there is a site associated with the person directly, it is used.
- if no site is found on the person, it looks at all groups to which the user belongs. If any of those groups have a site associated with it, the first one found is used.
- if no sites are found on the person or their groups, it looks at all roles the person is assigned. If any of those roles have a site associated with it, the first one found is used.

Add the MX_SITE_PREFERENCE variable to the Collaboration Server’s initialization file (bos.ini or ematrix.ini). This adjustment overrides the setting in the person, group, or role definition for the site preference, and should be set to the site that is local to the server. This ensures optimum performance of file checkin and checkout for Web clients.

Hidden Clause

You can use the hidden clause in an add group or add role command so that it does not appear in the User Chooser in eMatrix. Users who are aware of the hidden object’s existence can enter its name manually where appropriate. Hidden objects are listed with the MQL list command.

Property Clause

Integrators can assign ad hoc attributes, called properties, to the group or role. Properties allow associations to exist between administrative definitions that aren't already associated. The property information can include a name, an arbitrary string value, and a reference to another administration object. The property name is always required. The value string and object reference are both optional. The property name can be reused for different object references, that is, the name joined with the object reference must be unique for any object that has properties.

```
add role NAME property NAME [to ADMINTYPE NAME] [value STRING];
```

For more information, refer to *Working With Administration Properties* in Chapter 24.

Modifying a Group or Role Definition

After you establish a group or role definition, you can add or remove defining values. Refer to the description of [Modify Statement](#) in Chapter 1.

When modifying a group or role, it is important to consider how accesses are shared between parent and children. If you do not want a child to assume the same accesses to business objects as the parent, you will have to modify the policy so the privileges for the child role are specified. If a role is not specifically referenced in a policy, eMatrix will look for access hierarchically. For example, if the role has a parent and the parent is named in the policy, the child shares the parent's privileges.

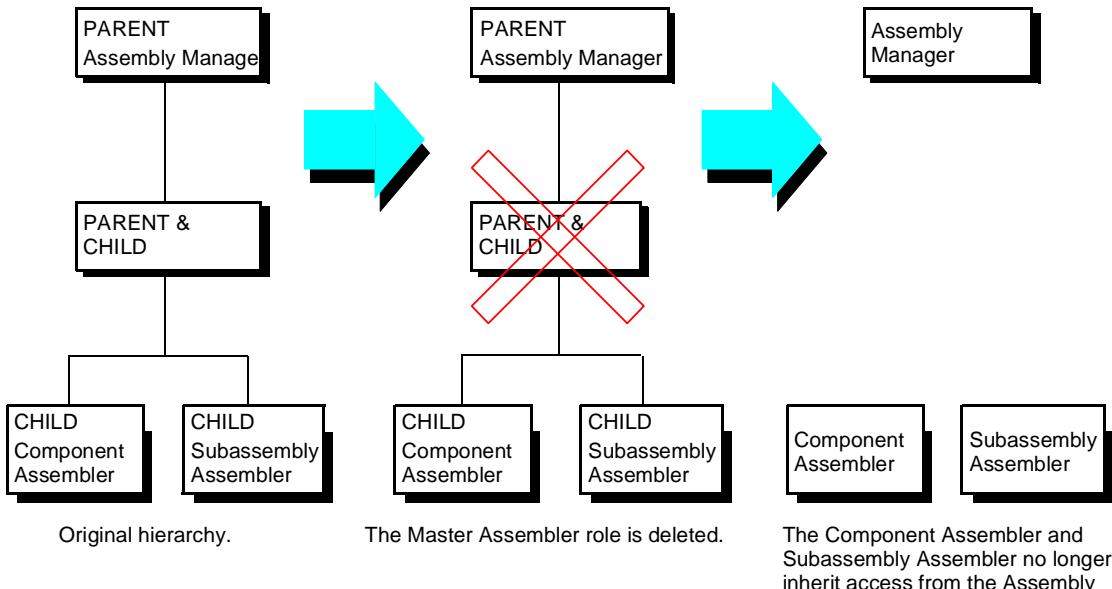
If you remove a parent or child, you might inadvertently remove access privileges from the children. Make sure that you consult the policies you have created when you alter the hierarchy of defined groups and roles.

Deleting a Group or Role Definition

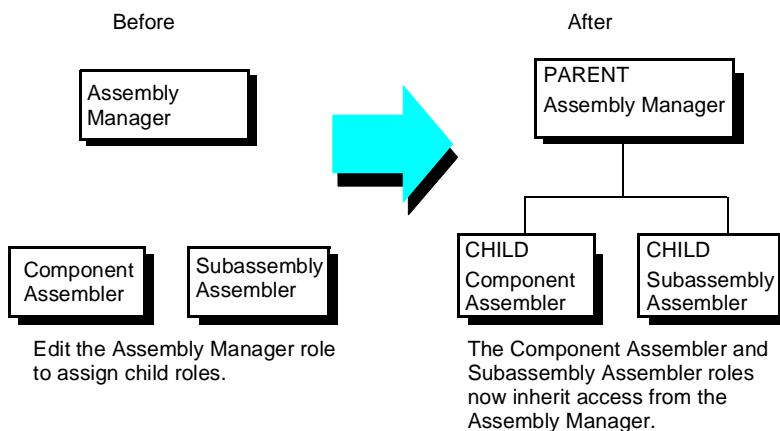
If a group or role is no longer required, you can delete it. Refer to the description in [Delete Statement](#) in Chapter 1.

When deleting a group or role, the elimination of linkages may affect another group's or role's access to business objects. For example, suppose you have four roles: Assembly Manager, Master Assembler, Component Assembler, and Subassembly Assembler. Assembly Manager is the parent of Master Assembler which is the parent of both Component Assembler and Subassembly Assembler (see the figure below).

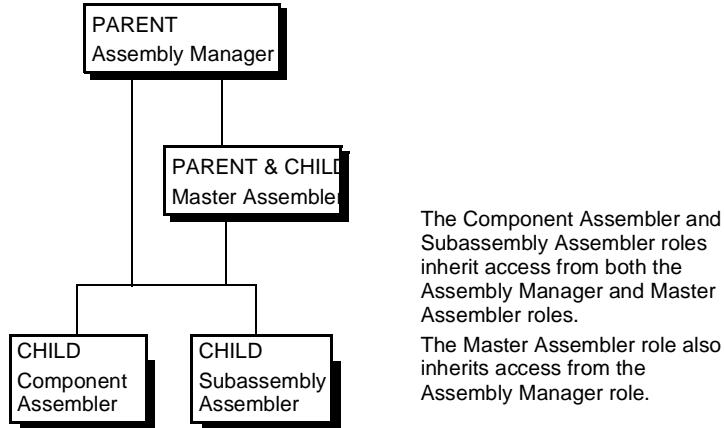
According to the Manufacturing policy, a role of Assembly Manager has read access to all business objects of type Assembly. Since all child roles of Assembler inherit the access abilities of their parent (unless specified otherwise), the Master Assembler, Component Assembler, and Subassembly Assembler roles also have read access. But what if you delete the Master Assembler role?



If the Master Assembler role is deleted, the linkages disappear between Master Assembler and the other roles. Component Assembler and Subassembly Assembler become stand-alone roles and lose the read access they inherited from the Master Assembler role. If this access was critical to performing their jobs, you can re-establish the linkages by modifying the role definition for Master Assembler. Simply define Master Assembler as having two child roles named Component Assembler and Subassembly Assembler (see the figure below).



You could also define both Assembly Manager and Master Assembler as parents of both Component Assembler and Subassembly Manager. Then if either of the roles is eliminated, the children still retain privileges inherited from the other.



Working with Associations

If users from a combination of different groups and roles will need access to a business object or set of objects, you should create an association.

Associations allow defining signature definitions such as “a Manager in the Products group AND a member of the Engineering group OR a member of the Design group OR a Vice-President.”

Uses for Associations

Signature Definitions

Suppose you are trying to define signature requirements of a policy that would govern business objects of type “Software Development.” Suppose the Approve signature definition for promotion of business objects from the state “Design” to the state “Implement” requires that the person must be a Project Leader and member of the Kernel Engineering group.

Without an association definition, you would have to hard code names of individuals who are Project Leaders and members of the Kernel Engineering group. However, if the number of individuals with such group/role assignments is large, you would have to manually enter all such names, which could be error prone. Also this signature definition would have to be maintained as group/role assignments change.

With associations, the following could be defined:

Project Leader and Kernel Engineering

With this association definition, the following individuals would have signature authority:

- All individuals who are assigned a role of Project Leader and belong to the Kernel Engineering group.

Another example of an association definition:

Designer or Project Leader and Kernel Engineering or Management

With this association definition, the following individuals would have signature authority:

- All individuals who are assigned a role of Designer.
- All individuals who are assigned a role of Project Leader who also belong to the Kernel Engineering group.
- All individuals who belong to a group called Management.

Notify

The Notification facility sends messages to a group of people when a business object enters a new state.

Suppose you want to remind all Quality Assurance people associated with the testing of Assembly 101 that the customer requested an extra test.

You could accomplish this by notifying the entire Quality Assurance Group, but the message would unnecessarily go out to people who did not work on Assembly 101.

You could also accomplish this by notifying the entire Assembly 101 Group, but the message would unnecessarily go out to people who do marketing, documentation, etc. for Assembly 101.

The association feature allows you to more effectively control the recipients of the notification message. You can send the message to only the desired set of people by using the following association definition: Quality Assurance and Assembly 101. The only people who receive the message are those in both the Quality Assurance and Assembly 101 Groups.

Defining an Association

Associations are created and defined with the MQL Add Association statement:

```
add association NAME [ADD_ITEM {ADD_ITEM}];
```

NAME is the name of the association you are creating. All associations must have a name assigned. When you create an association, you should assign a name that has meaning to both you and the user. This name cannot be shared with any other user types (groups, roles, persons, associations). The association name is limited to 127 characters. For additional information, refer to *Business Object Name* in Chapter 34.

After the name is defined, it will appear whenever association names are listed. For example, each of the following is a valid association name:

Managers and Administrators for Project A
Admin but Not HR
Final Review Team
QA and Help Desk Personnel

ADD_ITEM is an Add Association clause that provides more information about the association you are creating. While none of the clauses are required to make the association usable, they are used to define the association's relationship to existing groups, roles, and associations, as well as provide useful information about the association. The Add Association clauses are:

description VALUE
icon FILENAME
definition DEF_ITEM
[! not]hidden
property NAME [to ADMINTYPE NAME] [value STRING]

Each clause and the arguments they use are discussed in the sections that follow.

Description Clause

The Description clause of the Add Association statement provides general information for you and the user about the function of this association. There may be subtle differences between some associations. The Description clause enables you to point out the differences to the user.

There is no limit to the number of characters you can include in the description. However, keep in mind that the description is displayed when the mouse pointer stops over the

association in the User chooser. Although you are not required to provide a description, this information is helpful when a choice is requested.

For example, you could have two associations. Each association consists of one group and one role. The group contained in both associations is the same, but the roles are different. The group consists of all testers in the Quality Assurance department. One role tests parts before they are built into the product. The other role tests and reports on how well the parts function as part of the product. The following descriptions distinguish the two groups:

```
add association "Parts Testers"
description "Quality Assurance Team Testers, inspectors, for pre-assembly";
add association "Product Testers"
description "Quality Assurance Team Product Testers, inspectors, for beta test";
```

Icon Clause

Icons help users locate and recognize items by associating a special image with an association, such as simplified pictures of the associations you are defining. You can assign a special icon to the new association or use the default icon. The default icon is used when in view-by-icon mode. Any special icon you assign is used when in view-by-image mode. When assigning a unique icon, you must use a GIF image file. Refer to [Icon Clause](#) in Chapter 1 for a complete description of the Icon clause.

GIF filenames should not include the @ sign, as that is used internally by eMatrix.

Definition Clause

The definition clause of the Add Association statement defines which group(s), role(s), and association(s) are included in the new association. The definition must be enclosed in quotes.

```
definition "USER_ITEM [ {OPERATOR_ITEM USER_ITEM} ]"
```

USER_ITEM is the previously defined name of a group, role or association. An association definition can consist of all possible combinations of group(s), NOT-group(s), role(s), NOT-role(s), association(s), and NOT- association(s). Any USER_ITEM containing embedded spaces must be enclosed in quotes. Since the entire definition must also be enclosed in quotes, use single quotes for the USER_ITEM components and double quotes for the entire definition, or vice versa.

OPERATOR_ITEM can be either **&&** (to represent AND) or **||** (to represent OR).

eMatrix checks the association definition before accepting it as a valid definition. AND statements are evaluated before OR statements (that is, AND has a higher order of precedence). This order cannot be changed. The operator OR signifies the end of one expression and the beginning of another. For example, the following definition:

Admin and Production or Manager and Services

is evaluated as:

Admin and Production	or	Manager and Services
----------------------	----	----------------------

To create a definition using the AND operator

The AND operator requires that the person be defined in each group, role, or association that you include in the AND definition. For example:

```
definition "Engineering && Management"
```

In order for a person to satisfy this definition, the person must be defined in *both* the Engineering Group *and* in the Management Role.

To create a definition using the OR operator

The OR operator allows you to include persons defined in any of the groups, roles, or associations included in the OR definition. For example:

```
definition "Engineering || 'Senior Management'"
```

In order for a person to satisfy this definition, the person must be defined in *either* the Engineering Group *or* in the Senior Management Role.

Notice that single quotes are used for the two word name “Senior Management,” and that there is no space between the name and either single quote. A space between the second single quote and the double quote is optional.

To create a definition using Not Equal

If you want to exclude certain role(s) or group(s) or association(s) from the definition, you can use the Not Equal operator in the definition. Place an exclamation point (!) in front of whichever group/role/association you want to exclude. Be sure there is no space between the exclamation point and the name of the group/role/association. For example:

```
definition "!Engineering && Management"
```

In order for a person to satisfy this definition, the person must be defined in the Management role, but *not* in the Engineering Group.

To create a definition using multiple operators

You can use any combination of role(s) and group(s) and association(s) using AND and OR operators and Equal and Not Equal.

You could, for example, create a definition such as “A member of the Products Group AND a member of the Engineering Group OR a member of the Design Group OR a Vice President but NOT a member of the Marketing Group.” For example:

```
definition "!Marketing && Products && Engineering || Design || 'Vice President'"
```

Hidden Clause

You can specify that the new association is “hidden” so that it does not appear in the User chooser in eMatrix. Users who are aware of the hidden association’s existence can enter its name manually where appropriate. Hidden objects are accessible through MQL.

Property Clause

Integrators can assign ad hoc attributes, called Properties, to the association. Properties allow associations to exist between administrative definitions that aren’t already associated. The property information can include a name, an arbitrary string value, and a

reference to another administration object. The property name is always required. The value string and object reference are both optional. The property name can be reused for different object references, that is, the name joined with the object reference must be unique for any object that has properties.

```
add association NAME property NAME [to ADMINTYPE NAME] [value STRING];
```

For more information, refer to *Working With Administration Properties* in Chapter 24.

Copying (Cloning) an Association Definition

After an association is defined, you can clone the definition with the Copy Association statement. This statement lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy association SRC_NAME DST_NAME [MOD_ITEM {MOD_ITEM}];
```

SRC_NAME is the name of the association definition (source) to copied.

DST_NAME is the name of the new definition (destination).

MOD_ITEMS are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

Modifying an Association Definition

After an association is defined, you can change the definition with the Modify Association statement. This statement lets you add or remove defining clauses and change the value of clause arguments:

```
modify association NAME [MOD_ITEM {MOD_ITEM}];
```

NAME is the name of the association you want to modify.

MOD_ITEM is the type of modification to make. Each is specified in a Modify Association clause, as listed in the following table. Note that you only need to specify fields to be modified.

Modify Association Clause	Specifies that...
name NEW_NAME	The current association name is changed to the new name.
description VALUE	The current description, if any, is changed to the value entered.
icon FILENAME	The image is changed to the new image in the file specified.
definition DEF_ITEM	The definition is changed to the new definition specified.
hidden	The hidden option is changed to specify that the object is hidden.

Modify Association Clause	Specifies that...
nohidden	The hidden option is changed to specify that the object is not hidden.
property NAME [to ADMINTYPE NAME] [value STRING]	The named property is modified.
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.

As you can see, each modification clause is related to the clauses and arguments that define the association.

For example, assume you want to alter the definition for the Drivers First Shift association to reflect the addition of a new group. You might write this statement:

```
modify association "Drivers First Shift"
    definition "RR#6 && 'West Side' && RR#15";
```

This statement redefines the groups that are included in the association.

When modifying an association:

- The roles, groups, or associations that you assign to the association must already be defined within the eMatrix database. If they are not, an error will display when you try to assign them.
- Remember that altering the group or role access affects all persons included in the association that contains those groups or roles. If it is a singular case of special access, you may want to assign that person to the policy directly or define a role that is exclusively used by the person in question.

Deleting an Association

If an association definition is no longer required, you can delete it with the Delete Association statement:

```
delete association NAME;
```

NAME is the name of the association to be deleted.

When this statement is processed, eMatrix searches the list of associations. If the name is not found, an error message is displayed. If the name is found, the association is deleted and any linkages to that association are dissolved. For example, to delete the association named “Tree Specialists,” enter the following MQL statement:

```
delete association "Tree Specialists";
```

After this statement is processed, the association is deleted and you receive an MQL prompt for another statement.

Role-Based Visuals

Visuals (Filters, Tips, Cues, Toolsets, Tables, Views) are generally defined by users for their own personal use. They serve to set up a user's workspace in a way that is comfortable and convenient. Visuals can be used for many purposes, including organizing, prioritizing tasks, providing reminders, or streamlining access to information. Each user can define Visuals in a way that is most helpful to that person.

Visuals can also be defined and shared among users who are assigned to a role. For example, every person belonging to the role Accountant can have access to the same Visuals. This not only makes the work environment easier for a person just joining the department, but also facilitates communication among persons within a group.

For example, suppose there is a role called Manager. At a weekly manager's phone conference, each person sitting in front of a computer can, with the click of a mouse, switch Visuals so that the whole group is looking at the same thing. One could then suggest a filter to view a particular subset of objects, or look at a table, or refer to "all objects highlighted in red...."

The person setting up Visuals to be shared must have Business Administrator privileges.

Sharing Visuals

There are two methods to share visuals so that members of a group, role or association can use the visuals:

- A Business Administrator can copy the visuals from one user (person, group, role or association) to another using MQL commands.
- A Business Administrator can use the `set workspace` statement to change the Workspace currently active so that the Workspace of some other User (person, group, role or association) is active, and then create visuals within that new context. See [Setting the Workspace](#) for details.

Copying Visuals

After Visuals are defined within a session context, users with Business Administrator privileges can copy the definition. If you don't have Business Administrator privileges, then you need to be defined in the group, role, or association in order to copy from it to yourself.

The Copy statement lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy VISUAL SRC_NAME DST_NAME [fromuser USER] [touser USER] [overwrite] [MOD_ITEM  
{MOD_ITEM}];
```

VISUAL can be any one of the following: filter, tip, cue, toolset, table, view.

SRC_NAME is the name of the visual definition (source) to copied.

DST_NAME is the name of the new definition (destination).

USER can be the name of a person, group, role, or association.

`Overwrite` replaces an existing visual (or member, in the case of views) in the destination user.

The order of the `fromuser`, `touser` and `overwrite` clauses is irrelevant, but `MOD_ITEMS`, if included, must come last.

`MOD_ITEMS` are modifications that you can make to the new definition. `MOD_ITEMS` vary depending on which visual you are copying. A complete list can be found in the sections that follow.

For example, you can copy a cue definition using the following statement:

```
copy cue RedHiLite RedHiLite fromuser purcell touser Engineer
```

This statement copies a Cue named RedHiLite from the person purcell to the role Engineer.

If an error occurs during the copying of a Visual, the database will be left intact.

Setting the Workspace

The `set workspace` command allows you to change the Workspace currently active in MQL so that the Workspace of some other User (person, group, role or association) is visible.

Users can change to the Workspace of groups, roles, and associations to which they belong. Business Administrators can change to the Workspace of any group, role, association, or person.

The syntax for this command is:

```
set workspace user USERNAME
```

This command affects the behavior of all commands to which Workspace objects are applicable, including:

- all commands specific to Workspace objects (for example, add/modify/delete filter)
- `expand bus` (affects what filters are used to control output)

`USERNAME` is the name of a person, group, role or association.

When users (other than Business Administrators) set their Workspace to that of a group, role, or association, they cannot use commands that modify the Workspace, that is, the `add`, `modify`, and `delete` commands for Workspace objects. This restriction enforces the rule that only Business Administrators are permitted to change the Workspace of a group, role or association.

Note that `set context user USERNAME` will change the context to that of `USERNAME` regardless of an earlier invocation of `set workspace`.

12

Working With Attributes

Assigning Attributes to Objects and Relationships.....	308
Assigning Attributes to Objects.....	308
Assigning Attributes to Relationships	308
Defining an Attribute	309
Type Clause.....	309
Default Clause	311
Description Clause.....	311
Icon Clause.....	311
Rule Clause	312
Range Clause	312
Trigger Clause	317
Multiline Clause	317
Hidden Clause	317
Property Clause	317
Checking an Attribute	319
Copying and/or Modifying an Attribute Definition.....	320
Copying (Cloning) an Attribute Definition.....	320
Modifying an Attribute Definition	320
Deleting an Attribute	322

Assigning Attributes to Objects and Relationships

An *attribute* is any characteristic that can be assigned to an object or relationship. Objects may have attributes such as size, shape, weight, color, materials, age, texture, and so on.

You must be a Business Administrator to add or modify attributes. (Refer also to your Business Modeler Guide.)

Assigning Attributes to Objects

In eMatrix, objects are referred to by type, name, and revision. Object types are defined by the attributes they contain. When an object is created, the user specifies the object type and then eMatrix prompts for values for that object instance.

In eMatrix, you assign an attribute as a characteristic of a type of object. For example, assume the object is clothing. It might have attributes such as article type (for example, pants, coat, dress, shirt, and so on), size, cost, color, fabric type, and washing instructions. Now assume you are creating a new article of clothing. When you create this object, eMatrix prompts you to provide values for each of the assigned attributes. You might assign values such as jacket, size 10, \$50, blue, wool, and dry clean.

The specific value for each attribute can be different for each object instance. However, all objects of the same type have the same attributes.

Assigning Attributes to Relationships

Like business objects, a relationship may or may not have attributes. Since a relationship defines a connection between two objects, it has attributes when there are characteristics that describe that connection.

For example, an assembly could be connected to its components with a relationship called, “component of,” which could have an attribute called, “quantity.” When the component and the assembly are connected, the user would be prompted for the quantity or number of times the component is used in the assembly.

The same attributes could apply to either the objects or the relationship. When an object requires additional values for the same attribute in different circumstances, it is easier to assign the attributes to the relationship. Also, determine whether the information has more meaning to users when it is associated with the objects or the relationship.

Defining an Attribute

Before types and relationships can be created, the attributes they contain must be defined as follows:

```
add attribute NAME [ADD_ITEM {ADD_ITEM} ];
```

NAME is the name you assign to the attribute. Attribute names must be unique. The attribute name is limited to 127 characters. For additional information, refer to *Business Object Name* in Chapter 34.

ADD_ITEM is an Add Attribute clause that provides additional information about the attribute you are defining. The Add Attribute clauses are:

type {date integer string real boolean}
default VALUE
description VALUE
icon FILENAME
rule NAME
range RANGE_ITEM
trigger modify PROG_TYPE PROG_NAME [input ARG_STRING]
[! not]multiline
[! not]hidden
property NAME [to ADMINTYPE NAME] [value STRING]

The Type clause is always required.

Each clause and the arguments they use are discussed in the sections that follow.

Type Clause

The Type clause of the Add Attribute statement is always required. It identifies the type of values the attribute will have. An attribute can assume five different types of values. When determining the attribute type, you can narrow the choices by deciding if the value is a number or a character string.

Type	Description
String	<p>One or more characters. These characters may be numbers, letters, or any special symbols (such as \$ % ^ * &). Although numbers can be part of a character string, you cannot perform arithmetic operations on them. To perform arithmetic operations, you need a numeric type (Integer or Real).</p> <p><i>String attributes (as well as description fields) have a limit of 2,048 KB. If you expect to handle more data in an attribute, consider re-designing the schema to store the data in a checked-in file instead.</i></p>
Boolean	A value of TRUE or FALSE.

Type	Description												
Real	A number expressed with a decimal point (for example, 5.4321). The range of values it can assume depends on the local system architecture. eMatrix supports both 32-bit or 64-bit floating point numbers. Since real values are stored in the name format of the local system architecture, the precision and range of values varies from architecture to architecture. To obtain the exact range for your system, see your system manual.												
Integer	<p>A whole number whose range is defined by the local architecture. eMatrix supports all CPU architectures with the exception of signed 8-bit integers. Depending on the system architecture, an attribute with the integer type may assume a value within the following ranges:</p> <table> <tbody> <tr> <td>unsigned integer</td> <td>8 bits</td> <td>0 to 255</td> </tr> <tr> <td>signed integer</td> <td>16 bits</td> <td>-32,768 to 32,767</td> </tr> <tr> <td>unsigned integer</td> <td>16 bits</td> <td>0 to 65,535</td> </tr> <tr> <td>signed integer</td> <td>32 bits</td> <td>-2,147,483,647 to 2,147,483,647</td> </tr> </tbody> </table>	unsigned integer	8 bits	0 to 255	signed integer	16 bits	-32,768 to 32,767	unsigned integer	16 bits	0 to 65,535	signed integer	32 bits	-2,147,483,647 to 2,147,483,647
unsigned integer	8 bits	0 to 255											
signed integer	16 bits	-32,768 to 32,767											
unsigned integer	16 bits	0 to 65,535											
signed integer	32 bits	-2,147,483,647 to 2,147,483,647											
Date and Time	<p>Any collection of numbers or reserved words that can be translated into an expression of time. Times and dates can be expressed using the formats listed below. This includes the year, month, day, hour, minute, and/or second. Abbreviations or full-words are acceptable for the day of the week and month.</p> <p>In the following example, the day of the week is optional. Wed Feb 15, 1999 Another way to enter this date is: 2/15/99 The time of day. In the following example, the meridian and time zone information are optional: 01:30:00 PM EST Another example is: 13:30:00 When you enter both the date and time, the time should follow the date. For example: February 1, 1999 12:52:30 GMT The actual date/time is calculated based on the current time and date obtained from your system clock. The date range is January 1, 1902 to December 31, 2037. Consult the eMatrix <i>Installation Guide</i> "Configuring Date and Time Formats" section for details.</p>												

Once a type is assigned to an attribute, it cannot be changed. The user can associate only values of that type with the attribute.

For any field with a date format, even if you have a date setting in your initialization file (matrix.ini or ematrix.ini) and you input any date without a year, the date is accepted and converted to Sat Jan 01, 0000, 12:00:00 AM. Dates should be input as the full date, including the year.

Default Clause

The Default clause of the Add Attribute statement defines a default value for the attribute.

When a business object is created and the user does not fill in the attribute field, the default value is assigned. When assigning a default value, the value you give must agree with the attribute type. If the attribute should contain an integer value, you should not assign a string value as the default.

For example, assume you want to define an attribute called PAPER_LENGTH. Since this attribute will specify the size of a sheet of paper, you defined the type as an integer. For the default value, you might specify 11 inches or 14 inches, depending on whether standard or legal size paper is more commonly used.

As another example, assume you are defining an attribute called Label Color. If the most common color is yellow, you might define the attribute with this statement:

```
add attribute "Label Color"  
    default "yellow", type string;
```

If the user does not assign a value for the Label Color, “yellow” is assigned by default.

Description Clause

The Description clause of the Add Attribute statement provides general information for you and the user about the function of the attribute. There may be subtle differences between attributes; the Description clause points out the differences to the user.

The description can consist of a prompt, comment, or qualifying phrase. There is no limit to the number of characters you can include in the description. However, keep in mind that the description is displayed when the mouse pointer stops over the attribute in the Attribute chooser. Although you are not required to provide a description, this information is helpful when a choice is requested.

Although a Description clause is not required, it can clarify an expected value or the information presented when editing or viewing the attribute. For example, when defining an attribute named Label, you might use one of these Description clauses:

```
add attribute Label  
    description "Shipping Destination Label"  
    type string;  
  
add attribute Label  
    description "Is ship-to label required?"  
    type string;  
  
add attribute Label  
    description "Enter the shipping label number"  
    type integer;
```

The information assigned to the Label attribute differs significantly in these statements. In the first statement, the attribute might be a shipping address. In the second statement, it might have a value of yes, no, or unknown. In the third statement, a number is required.

Icon Clause

Icons help users locate and recognize items by associating a special image with the attribute. You can assign a special icon to the new attribute or use the default icon. The default icon is used when in view-by-icon mode. Any special icon you assign is used when in view-by-image mode. When assigning a unique icon, you must use a GIF image file. Refer to [Icon Clause](#) in Chapter 1 for a complete description of the Icon clause.

Rule Clause

Rules are administrative objects that define specific privileges for various eMatrix users. The Rule clause of the Add Attribute statement enables you to specify an access rule to be used for the attribute.

```
add attribute NAME rule RULENAME;
```

Range Clause

The Range clause of the Add Attribute statement defines the range of values the attribute can assume. This range provides a level of error detection and gives the user a way of searching a list of attribute values. If you define an attribute as having a specific range, any value the user tries to assign to that attribute is checked to determine if it is within that range. Only values within the defined range are allowed.

When writing a Range clause, use one of the following forms depending on the types and amount of range values:

range RELATIONAL_OPERATOR VALUE
range PATTERN_OPERATOR PATTERN
range between VALUE {inclusive exclusive} VALUE {inclusive exclusive}
range program PROG_NAME [input ARG_STRING]

If you have an attribute with a default value that is outside the ranges defined and you try to create a business object without changing the attribute value, eMatrix does not check the default value against the ranges. A trigger check could be written on the attribute to force a user to enter a value.

Each form is described below. The method and clauses you use to define a range are a matter of preference. Select the clauses that make the most sense to you and then test the range to be sure it includes only valid values.

Range Compared with a Relational Operator

This form offers greater flexibility in defining the range of possible values:

range RELATIONAL_OPERATOR VALUE

A relational operator compares the user's value to a set of possible values. Choose from these relational operators:

Operator	Operator Name	Function
=	is equal to	The user value must equal this value or another specified valid value (given in another Range clause). When comparing user-supplied character values to the range value, uppercase and lowercase letters are equivalent.
!=	is not equal to	The user-supplied value must not match the value given in this clause. If it matches, the user is notified that the value is invalid. When comparing user-supplied character values to the range value, uppercase and lowercase letters are equivalent.
<	is less than	The user value must be less than the given range value. If the user value is equal to or greater than the range value, it is not allowed.
<=	is less than or equal to	The user value must be less than or equal to the given range value. If the user value is greater than the range value, it is not allowed.
>	is greater than	The user value must be greater than the given range value. If the user value is equal to or less than the range value, it is not allowed.
>=	is greater than or equal to	The user value must be greater than or equal to the given range value. If the user value is less than the range value, it is not allowed.

Depending on the relational operator you use, you can define a range set that is very large, or a set that contains a single value. When you use a relational operator, the value provided by the user is compared with the range defining value. If the comparison is true, the value is allowed and is assigned to the attribute. If the comparison is not true, the value is considered invalid and is not allowed.

For example, assume you want to restrict the user to entering only positive numbers. In this case, you could define the range using either of the following clauses:

```
range > -1
Or:
range >= 0
```

If the user enters a negative number (such as -1), these statements are false (-1 is not greater than -1 and is not greater than or equal to zero). Therefore the value is invalid.

You may have an attribute with a few commonly entered values but that can actually be any value. To provide the user with the ability to select the commonly entered values from a menu, but also allow entry of any value, you would:

- Add the ranges of **Equal** values for the common values.
- Add a range of **Not Equal** to xxxxx.

This will allow any value (except xxxxx) and also provide a list from which to choose the common values.

When defining ranges for character strings, remember that you can also perform comparisons on them. By using the ASCII values for the characters, you can determine

whether a character string has a higher or lower value than another character string. For example “Boy” is less than “boy” because uppercase letters are less than lowercase letters and “5boys” is less than “Boy” because numbers are less than uppercase letters. For more information on the ASCII values, refer to an ASCII table.

But what would you do if the attribute value had a second part that was to start with the letters REV? The next form of the Range clause is available for this reason.

Range Compared with a Pattern (Special Character String)

This form, used exclusively for character strings, uses a special character string called a *pattern*.

range PATTERN_OPERATOR PATTERN

Patterns are powerful tools for defining ranges because they can include *wildcard* characters. Wildcard characters can be used to represent a single digit or a group of characters. This allows you to define large ranges of valid values.

For example, you can define an attribute’s range as “DR* REV*” where the asterisk (*) is a wildcard representing *any* character(s). This range allows the user to enter any value, as long as the first half begins with the letters “DR” and the second half begins with the letters “REV”.

When using a pattern to define a range, you must use a pattern operator. These operators compare the user’s value with the pattern range. All the pattern operators allow for wildcard comparisons. Two of them check the user’s entry for an exact match (including checks for uppercase and lowercase).

Operator	Operator Name	Function
match	match	The user’s character string must match the exact pattern value given, including uppercase and lowercase letters. For example, “Red Robin” is not a sensitive match for the pattern value “re*ro*” since the uppercase R’s do not match the pattern.
!match	not match	The user’s character string must not match the exact pattern value. For example, if the user enters “Red” when the pattern value is “red”, the value is allowed; “Red” is not an exact match to “red”.
smatch	string match	The user’s character string must match only the general pattern value, independent of case. Case is ignored so that “RED” is considered a match for “red”.
!smatch	not string match	The user’s character string must not match the general pattern value, independent of case. For example, assume the range pattern is defined as “re*ro*”. The value “Red Robin” is not allowed, although “red ribbon” is allowed. That is because the first value is a pattern match (regardless of case difference) and the second is not.

When the user enters a character string value, eMatrix uses these operators to compare that value to the defined range pattern. If the comparison results in a true value, the user value is considered valid and is assigned to the attribute. If the comparison is false, the user value is considered out of range and is not valid.

Multiple Ranges

When defining the range, remember that you can use more than one Range clause. More than one clause may be required.

```
range between VALUE {inclusive|exclusive} VALUE {inclusive|exclusive}
```

For example, if you have a situation where the attribute value can be any uppercase letter except for I or O (since they can be confused with numbers), you could define the range as:

```
range >= A range <= Z range != I range != O
```

This is the same as:

```
range between A inclusive Z inclusive range != I range != O
```

Ranges Defined Within a Program

eMatrix allows you to use a program to define range values. This allows the flexibility to change range values depending on conditions. Use the range program clause to specify the name of the program to execute. For example:

```
range program SetRanges;
```

In this example, the name of the program to execute is SetRanges.

You can define arguments to be passed into the program. Your program could change the attribute range depending on the argument passed. For example:

```
range program SetRanges attrang1;
```

When you pass arguments into the program they are referenced by variables within the program. Variables 0, 1, 2... etc. are reserved by the system for passing in arguments.

Environment variable “0” always holds the program name and is set automatically by the system.

Arguments following the program name are set in environment variables “1”, “2”, ... etc.

Programs should return the list of choices in the variable that has the same name as the program name, which can be obtained through argument “0.” Be sure to use the `global` keyword on your `set env` command when passing the list back.

See [Runtime Program Environment](#) in Chapter 18 for additional information on program environment variables.

The following is output from a MQL session. The attribute type ‘designerName’ produces the choices ‘Tom,’ ‘Dick,’ ‘Harry,’ ‘Larry,’ ‘Curly,’ and ‘Moe.’ Note that an attribute type can have any number of ranges, but only one range can be of type program.

```
MQL<1>print attr designerName;
attribute designerName
  type string
  description
  default Tom
  range = Tom
  range = Dick
  range = Harry
  range uses program nameRange
```

```

    not multiline
MQL<2>print prog nameRange;
program nameRange
  mql
  description
  code 'tcl;
eval {
  # set the event
  set event [mql get env EVENT]
  # set the choices
  set names {Larry Curly Moe}
  # set the output variable (arg 0 is this program's name)
  set output [mql get env 0]
# test event, and either generate choices, or test value
if { $event == "attribute choices" } {
  # note that choices are returned in a global RPE variable
  mql set env global $output $names
} else {
  # assume that it is safe to unset global RPE variable during value test
  mql unset env global $output
  # set the value
  set value [mql get env ATTRVALUE]
  # test the value
  if {[lsearch -exact $names $value] == -1} {
    # value not in list, return non-zero
    exit 1
  } else {
    # value in list, return zero
    exit 0
  }
}
}

```

The following macros are available to Range Programs:

- **Owning Item information macros.** There are 3 scenarios:
 - a)** If the attribute is “owned” by a business object, the macros available are OBJECT, TYPE, NAME, and REVISION.
 - b)** If a relationship instance “owns” the attribute, the RELID macro is provided.
 - c)** During query formulation the owner is unknown so no owner macros are available.
- **Invocation Information.** INVOCATION will always equal “range”.
- **Event information.** EVENT and possibly ATTRVALUE. For example:
 - a)** When the range program is being asked to produce all legal values, EVENT will equal “attribute choices”.
 - b)** When the range program is being asked to check the legality of a given value, EVENT will equal “attribute check” and the ATTRVALUE macro will also be provided.
- **Attribute information.** ATTRNAME and ATTRTYPE. For example:
 ATTRNAME=designerName
 ATTRTYPE=String
- **Basic information.** VAULT, USER, TRANSACTION, HOST, APPLICATION, and LANGUAGE.

Trigger Clause

Event Triggers provide a way to customize eMatrix behavior through Program Objects. Triggers can contain up to three Programs, (a check, an override, and an action program) which can all work together, or each work alone. Attributes support the use of triggers on the modify event. For more information on Event Triggers, refer to [Overview of Event Triggers](#) in Chapter 20. The syntax for the trigger clause is::

```
trigger modify {action|check|override} PROG_NAME [input ARG_STRING];
```

For example, to assign a check trigger called OnApprove, you would use:

```
trigger modify check OnApprove;
```

In this example, the name of the program to execute is OnApprove.

You can define arguments to be passed into the program. For example:

```
trigger modify check OnApprove input ChngChk;
```

In this example, the argument passed into the OnApprove program is ChngChk.

When you pass arguments into the program they are referenced by variables within the program. Variables 0, 1, 2... etc. are reserved by the system for passing in arguments.

Environment variable “0” always holds the program name and is set automatically by the system.

Arguments following the program name are set in environment variables “1”, “2”,... etc.

See [Using the Runtime Program Environment](#) in Chapter 19 for additional information on program environment variables.

Multiline Clause

If you define the attribute type value as “string,” you can specify the format of the data entry field. Use the `multiline` clause if you want the data entry field to consist of multiple lines. If this clause is specified, the text wraps to the next line as the user types. The text box is scrollable.

If `multiline` is not specified or if `!multiline` (`notmultiline`) is specified, then the data entry field consists of a single line. If the amount of text exceeds the size of the field shown, the line of text scrolls to the left as the user is typing.

Hidden Clause

You can specify that the new attribute is “hidden” so that it does not appear in the Attribute chooser or in the list of attributes for an object in eMatrix. You may want to use the hidden option if, for example, an object is under development or if it is intended only for your personal use. Hidden objects are accessible through MQL.

In desktop eMatrix Navigator, hidden attributes are displayed only in the Object Inspector. In the Web version, hidden attributes do not appear at all.

Property Clause

Integrators can assign ad hoc attributes, called Properties, to the attribute. Properties allow associations to exist between administrative definitions that aren’t already associated. The

property information can include a name, an arbitrary string value, and a reference to another administration object. The property name is always required. The value string and object reference are both optional. The property name can be reused for different object references, that is, the name joined with the object reference must be unique for any object that has properties.

```
add attr NAME  
    property NAME [to ADMINTYPE NAME] [value STRING];
```

For additional information on properties, see *Overview of Administration Properties* in Chapter 24.

Checking an Attribute

Any attribute can be checked to find out whether a specified value is within the given range for that attribute:

```
check attribute NAME VALUE [businessobject TYPE NAME REV] [connection RELID];
```

NAME is the name of the attribute.

VALUE is the value you want to check.

Because an attribute's Range Program can produce different legal values depending on the state or settings of the owning business object or relationship instance, the business object TYPE NAME REV or RELID can be used to specify the owner of the attribute:

RELID is the id of the relationship instance that owns the attribute.

For example, to find out if "3" is a legal value for an attribute called "Priority," you could issue the following command:

```
check attribute Priority 3;
```

If it is not a legal value, the following is returned:

```
'3' violates range for attribute 'Priority'
```

Copying and/or Modifying an Attribute Definition

Copying (Cloning) an Attribute Definition

After an attribute is defined, you can clone the definition with the Copy Attribute statement. This statement lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy attribute SRC_NAME DST_NAME [MOD_ITEM {MOD_ITEM}];
```

SRC_NAME is the name of the attribute definition (source) to copied.

DST_NAME is the name of the new definition (destination).

MOD_ITEMS are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

Modifying an Attribute Definition

After an attribute is defined, you can change the definition with the Modify Attribute statement. This statement lets you add or remove defining clauses and change the value of clause arguments:

```
modify attribute NAME [MOD_ITEM {MOD_ITEM}];
```

NAME is the name of the attribute you want to modify.

MOD_ITEM is the type of modification you want to make.

You cannot alter the Type clause. If you must change the attribute type, delete the attribute and add it again using a new Type clause.

You can make the following modifications to an existing attribute definition. Each is specified in a Modify Attribute clause, as listed in the following table. Note that you only need to specify fields to be modified.

Modify Attribute Clause	Specifies that...
name NEW_NAME	The current attribute name changes to the new name entered.
default VALUE	The current default value, if any, is set to the value entered.
description VALUE	The current description value, if any, is set to the value entered.
icon FILENAME	The image is changed to the new image in the file specified.
add range RANGE_ITEM	The given Range clause is added to the existing list of Range clauses. This Range clause must obey the same construction and syntax rules as when defining an attribute.
remove range RANGE_ITEM	The given Range clause is removed from the existing list of Range clauses. The given Range clause must exist, or an error message is displayed.

Modify Attribute Clause	Specifies that...
add trigger modify PROG_TYPE PROG_NAME	The current trigger program name is changed to the new name. PROG_TYPE is the type of trigger program (check, override, or action) and PROG_NAME is the name of the program that replaces the current trigger program.
remove trigger modify PROG_TYPE	The specified trigger program is removed. PROG_TYPE is the type of trigger program (check, override, or action).
add rule NAME	The named access rule is added.
remove rule NAME	The named access rule is removed
hidden	The hidden option is changed to specify that the object is hidden.
nohidden	The hidden option is changed to specify that the object is not hidden.
property NAME [to ADMINTYPE NAME] [value STRING]	The named property is modified.
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.

As you can see, each modification clause is related to the clauses and arguments that define the attribute. For example, you would use the Default clause of the Modify Attribute statement to modify the Default clause that you used in the Add Attribute statement.

Assume you have the following attribute definition:

```
attribute Units
  description "Measuring units"
  type string
  default "Linear Feet"
  range = Quarts
  range = "Linear Feet";
```

Now you have decided to use the attribute for measuring distances only. While it might be faster to define a new attribute, you could modify this attribute with the following statement:

```
modify attribute Units
  description "Units for Measuring Distances"
  remove range = Quarts
  add range = "Linear Yards";
```

After this statement is processed, the attribute definition for the Units attribute is:

```
attribute Units
  description "Units for Measuring Distances"
  type string
  default "Linear Feet"
  range = "Linear Feet"
  range = "Linear Yards";
```

Deleting an Attribute

If an attribute is no longer required, you can delete it by using the Delete Attribute statement:

```
delete attribute NAME;
```

NAME is the name of the attribute to be deleted.

When this statement is processed, eMatrix searches the list of defined attributes. If the name is found, that attribute is deleted. If the name is not found, an error message is displayed.

For example, to delete the Shipping Address attribute and the Label attribute, enter the following two statements:

```
delete attribute "Shipping Address";  
delete attribute Label;
```

After these statements are processed, the attributes are deleted and you receive an MQL prompt for another statement.

13

Working With Types

Type Defined	324
Type Characteristics.....	325
Implicit and Explicit	325
Inherited Properties	325
Defining a Type	326
Description Clause.....	326
Icon Clause.....	327
Attribute Clause	327
Derived Clause	328
Abstract Clause	329
Method Clause..	329
Form Clause	329
Trigger Clause	330
Hidden Clause	330
Property Clause	331
Copying and/or Modifying a Type Definition.....	332
Copying (Cloning) a Type Definition.....	332
Modifying a Type Definition	332
Three Base Types	334
Localizing Base Types.....	334
Deleting a Type	335

Type Defined

A *type* identifies a kind of business object and the collection of attributes that characterize it. When a type is created, it is linked to the (previously-defined) attributes that characterize it. It may also have methods and/or triggers associated (refer to [Working With Programs](#) and [Working With Event Triggers](#) for more information). Types are defined by the Business Administrator and are used by eMatrix users to create business object instances.

A type can be *derived* from another type. This signifies that the derived type is of the same kind as its parent. For example, a Book is a kind of Publication which in turn is a kind of Document. In this case, there may be several other types of Publications such as Newspaper, Periodical, and Magazine.

This arrangement of derived types is called a *type hierarchy*. Derived types share characteristics with their parent and siblings. This is called *attribute inheritance*. When creating a derived type, other attributes, methods, and triggers can be associated with it, in addition to the inherited ones. For example, all Periodicals may have the attribute of Page Count. This attribute is shared by all Publications and perhaps by all Documents. In addition, Periodicals, Newspapers, and Magazines might have the attribute Publication Frequency.

You must be a Business Administrators to add or modify types. (Refer also to your Business Modeler Guide.)

Type Characteristics

Implicit and Explicit

Types use explicit and implicit characteristics:

- *Explicit characteristics* are attributes that you define and are known to eMatrix.
- *Implicit characteristics* are implied by the name only and are known only to the individual user.

For example, you may create a type called “Tax Form” which contains administrator-defined explicit attributes such as form number, form type, and tax year. Or, Tax Form may contain no explicit attributes at all.

When a type exists without administrator-defined attributes, it still has implicit characteristics associated with it. You would know a tax form when you saw it and would not confuse it with a type named “Health Form.” But the characteristics you use to make the judgment are implicit—known only by you and not eMatrix.

Inherited Properties

Types can inherit properties from other types. In eMatrix:

- *Abstract types* act as categories for other types.
Abstract types are not used to create any actual instances of the type. They are useful only in defining characteristics that are inherited by other object types. For example, you could create an abstract type called Income Tax Form. Two other abstract types, State Tax Form and Federal Tax Form, inherit from Income Tax Form.
- *Non-abstract types* are used to create instances of business objects.
With non-abstract types, you can create instances of the type. For example, assume that Federal Individual Tax Form is a non-abstract type. You can create business objects in eMatrix that contain the actual income tax forms for various individuals. One object might be for a person named Joe Smith and another one for Mary Jones. Both objects have the same type and characteristics although the contents are different based on the individuals.

Defining a Type

An object type is created with the Add Type statement:

```
add type NAME [ADD_ITEM {ADD_ITEM}];
```

NAME is the name you assign to the type. The type name is limited to 127 characters. For additional information, refer to *Business Object Name* in Chapter 34.

ADD_ITEM is an Add Type clause which provides additional information about the type:

description VALUE
icon FILENAME
attribute NAME {, NAME}
derived TYPE_NAME
abstract [true false]
method PROG_NAME {, PROG_NAME}
form NAME {,NAME}
trigger EVENT_TYPE {action check override} PROG_NAME [input ARG_STRING]
[! not]hidden
property NAME [to ADMINTYPE NAME] [value STRING]

All these clauses are optional. You can define a type by simply assigning a name to it. If you do, eMatrix assumes that the type is non-abstract, uses the default type icon, does not contain any explicit attributes, and does not inherit from any other types. If you do not want these default values, add clauses as necessary. You will learn more about each Add Type clause in the sections that follow.

Performance problems may be noticed when adding a Type to a very large type hierarchy. If you experience this, from Oracle, turn off “cost-based optimization” and the situation should improve. Refer to Oracle documentation for more information.

Description Clause

The Description clause of the Add Type statement provides general information for you and the user about the function of the type you are defining. There may be subtle differences between types; the Description clause enables you to point out the differences to the user.

There is no limit to the number of characters you can include in the description. However, keep in mind that the description is displayed when the mouse pointer stops over the type in the Type chooser. Although you are not required to provide a description, this information is helpful when a choice is requested.

If an object is assigned the wrong type, it may be inaccessible to the users or restricted from desired connections.

The Description clause may consist of a prompt, comment, or qualifying phrase. For example, if you are defining a type named Drawing, you might write this statement:

```
add type Drawing description "2-D illustration";
```

This Description clause provides information to you and the user about the kinds of files and information associated with this type. As another example, you might define a type named “Federal Income Tax Form” with the following Description clause. It provides qualifying information about which forms should use this type.

```
add type "Federal Income Tax Form"  
description "Use for individual income tax forms 401 and  
401A, only";
```

Icon Clause

Icons help users locate and recognize items by associating a special image with the type. You can assign a special icon to the new type or use the default icon. The default icon is used when in view-by-icon mode. Any special icon you assign is used when in view-by-image mode. When assigning a unique icon, you must use a GIF image file. Refer to [Icon Clause](#) in Chapter 1 for a complete description of the Icon clause.

GIF filenames should not include the @ sign, as that is used internally by eMatrix.

Attribute Clause

The Attribute clause of the Add Type statement assigns explicit attributes to the type. These attributes must be previously defined with the Add Attribute statement. (See [Defining an Attribute](#) in Chapter 12.) If they are not defined, an error message is displayed.

Adding an attribute to a business type should not be included in a transaction with other extensive operations, especially against a distributed database. This is a “special” administrative edit, in that it needs to update all business objects of that type with a default attribute.

For the eMatrix Navigator user, the display of attributes (when creating objects or viewing attributes) will appear in the reverse order of the programmed order. Therefore, you should put the attribute you want first last in the MQL script.

A type can have any combination of attributes associated with it. For example, the following Add Type statement assigns three attributes to the Shipping Form type:

```
add type "Shipping Form"  
description "Use for shipping materials to external locations"  
attribute "Label Type"  
attribute "Date Shipped"  
attribute "Destination Type";
```

Three explicit attributes are associated with this type definition: Label Type, Date Shipped, and Destination Type. When the user creates a business object of a Shipping

Form type, s/he will be required to provide values for these attributes. For example, the user might enter the following values at the attribute prompts:

Prompt:	User Response:
Label Type	Overnight Express
Date Shipped	December 22, 1999
Destination Type	Continental U.S.

These values are then associated with that instance along with any files the user may want to check in.

Derived Clause

Use the Derived clause to identify an existing type as the parent of the type you are creating. The parent type can be abstract or non-abstract. A child type inherits the following items from the parent:

- all attributes
- all methods
- all triggers
- governing policies

For example, if two policies list the parent type as a governed type, then those two policies can also govern the child type. Note that in such a case, the child type is not listed as a governed type in the policy definitions.

- allowed relationships

For example, if a relationship allows the parent type to be on the from end, then the child type can also be on the from end of the relationship. The child type is not listed in the relationship definition.

Assigning a parent type is an efficient way to define several object types that are similar because you only have to define the common items for one type, the parent, instead of defining them for each type. The child type inherits all the items listed above from the parent but you can also add attributes, programs, and methods directly to the child type. Similarly, you can (and probably will) assign the child type to a policy or relationship that the parent is not assigned to. Any changes you make for the parent are also applied to the child type.

For example, suppose you have a type named “Person Record”, which includes four attributes: a person’s name, telephone number, home address, and social security number. Now, you create two new types named “Health Record” and “Employee Record.”

```
add type "Health Record"
      derived "Person Record";
add type "Employee Record"
      derived "Person Record";
```

Both new types require the attributes in the Person Record type. Rather than adding each of the four Person Record attributes to the new types, you can make Person Record the parent of the new types. The new types then inherit the four attributes, along with any methods, programs, policies, and relationships for parent.

Abstract Clause

An abstract type indicates that a user will not be able to create a business object of the type. An abstract type is helpful because you do not have to reenter groups of attributes that are often reused. If an additional field is required, it needs to be added only once. For example, the “Person Record” object type might include a person’s name, telephone number, home address, and social security number. While it is a commonly used set of attributes, it is unlikely that this information would appear on its own. Therefore, you might want to define this object type as an abstract type.

Since this type is abstract, there will never be any actual instances made of the Person Record type. However, it can be inherited by other object types that might require the attribute information. Even though a user may never be required to enter values for the attributes of a Person Record object, he may have to enter values for these attributes for an object that inherited the Person Record attributes (such as an Employee Record or Health Record).

Use one the following clauses:

```
abstract true  
Or:  
abstract false
```

If the user can create an object of the defined type, set the abstract argument to `false`. If the user cannot, set the abstract argument to `true`. If you do not use the Abstract clause, `false` is assumed, allowing users to create instances of the type.

For example, in the following definition of Federal Individual Income Tax Form, the user can create an object instance because an Abstract clause was not included in the definition:

```
add type "Federal Individual Income Tax Form"  
derived "Federal Tax Form";
```

This definition is equivalent to:

```
add type "Federal Individual Income Tax Form"  
derived "Federal Tax Form"  
abstract false;
```

Method Clause

The Method clause of the Add Type statement assigns a method to the type. A method is a program that can be executed from eMatrix when it is associated with the selected object. Programs selected as methods require a business object as a starting point for executing. For example, the following adds the existing program named “calculate tax” to the Federal Individual Income Tax Form.

```
add type "Federal Individual Income Tax Form"  
derived "Federal Tax Form"  
method "calculate tax";
```

A user in eMatrix could then select any Federal Individual Income Tax Form object and execute the program “calculate tax.”

Form Clause

The Form clause of the Add Type statement assigns a form design to the type. This form must be previously defined with the Add Form statement. (See *Defining a Form* in Chapter 23.) If it is not defined, an error message is displayed.

A type can have any number of forms associated with it. For example, the following Add Type statement assigns three forms to the Employee types:

```
add type "Employee Record"
description "employees of Acme, Inc."
form "insurance information"
form "work history"
form "pension and savings plans;"
```

Trigger Clause

Event Triggers provide a way to customize eMatrix behavior through Program objects. Triggers can contain up to three Programs, (a check, an override, and an action program) which can all work together, or each work alone. The Trigger clause specifies the program name, which event causes the trigger to execute, and which type of trigger program it is. Types support triggers for many events. For more information on Event Triggers, refer to [Overview of Event Triggers](#) in Chapter 20.

The format of the trigger clause is:

```
trigger EVENT_TYPE {action|check|override} PROG_NAME [input ARG_STRING{
```

EVENT_TYPE specifies the type of event that causes the trigger program to execute. It can be any of the following:

changevault	changename	changeowner
changepolicy	changetype	checkin
checkout	connect	copy
create	delete	disconnect
grant	lock	modifyattribute
modifydescription	removefile	revision
revoke	unlock	

PROG_NAME is the name of the program to execute when the event occurs.

ARG_STRING is a string of arguments to be passed into the program. When you pass arguments into the program they are referenced by variables within the program. Variables 0, 1, 2... etc. are reserved by the system for passing in arguments.

Environment variable “0” always holds the program name and is set automatically by the system.

Arguments following the program name are set in environment variables “1”, “2”,... etc.

See [Using the Runtime Program Environment](#) in Chapter 19 for additional information on program environment variables.

Hidden Clause

You can specify that the new type is “hidden” so that it does not appear in the Type chooser or in any dialogs that list types in eMatrix. You may want to use the hidden option if, for example, an object is under development or if it is intended only for your personal use. Users who are aware of the hidden type’s existence can enter its name manually where appropriate. Hidden objects are also accessible through MQL.

Property Clause

Integrators can assign ad hoc attributes, called Properties, to the type. Properties allow associations to exist between administrative definitions that aren't already associated. The property information can include a name, an arbitrary string value, and a reference to another administration object. The property name is always required. The value string and object reference are both optional. The property name can be reused for different object references, that is, the name joined with the object reference must be unique for any object that has properties.

```
add type NAME  
    property NAME [to ADMINTYPE NAME] [value STRING];
```

For additional information on properties, see *Overview of Administration Properties* in Chapter 24.

Copying and/or Modifying a Type Definition

Copying (Cloning) a Type Definition

After a type is defined, you can clone the definition with the Copy Type statement. This statement lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy type SRC_NAME DST_NAME [MOD_ITEM {MOD_ITEM}];
```

SRC_NAME is the name of the type definition (source) to copied.

DST_NAME is the name of the new definition (destination).

MOD_ITEMS are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

Modifying a Type Definition

After a type is defined, you can change the definition with the Modify Type statement. This statement lets you add or remove defining clauses and change the value of clause arguments:

```
modify type NAME [MOD_ITEM {MOD_ITEM}];
```

NAME is the name of the type you want to modify.

MOD_ITEM is the type of modification you want to make.

You can make the following modifications. Each is specified in a Modify Type clause, as listed in the following table. Note that you need to specify only fields to be modified.

Modify Type Clause	Specifies that...
name NEW_NAME	The current type name changes to that of the new name entered.
description VALUE	The current description, if any, changes to the value entered.
icon FILENAME	The image is changed to the new image in the file specified.
add attribute NAME	The named attribute is added to the type's list of explicit attributes.
add form NAME	The named form is added to the type.
add method PROG_NAME	The named method is added to the type.
add trigger EVENT_TYPE PROG_TYPE PROG_NAME	The named trigger program name is added. This Modify clause must obey the same construction and syntax rules as when defining a type.
remove attribute NAME	The named attribute is removed from the type's list of explicit attributes.
remove form NAME	The named form is removed from the type.
remove method PROG_NAME	The named method is removed from the type.

Modify Type Clause	Specifies that...
remove trigger EVENT_TYPE PROG_TYPE	The named trigger program is removed from the type. This clause must obey the same construction and syntax rules as when defining a type.
derived TYPE_NAME	The type being modified inherits attributes, methods, triggers, governing policies, and allowed relationships of the type named.
abstract true	Business object instances of this type cannot be created.
abstract false	Business object instances of this type can be created.
hidden	The hidden option is changed to specify that the object is hidden.
nohidden	The hidden option is changed to specify that the object is not hidden.
property NAME [to ADMINTYPE NAME] [value STRING]	The named property is modified.
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.

As you can see, each modification clause is related to the arguments that define the type. For example, the following statement changes the name and attribute list of the Shipping Form type:

```
modify type "Shipping Form" name "Shipping Label"
add attribute "Label Color";
```

Three Base Types

There are three base types that must be defined in order to use some basic eMatrix functions. These types are:

- Annotation
- Attachment
- Report

By creating types of these names and deriving new types from them, the number of types available under Annotation, Attachment, and Report can be kept to a manageable number. For example, when a new report is created in eMatrix, (by selecting New> Report from the object menu) a Report object type must be selected from the Type Chooser. Only the types derived from Report will be displayed, eliminating types that are not appropriate for a Report. The Attachment and Annotation types work the same way.

Note that applicable relationships and policies for these types must be defined as well, as described in [Chapter 14, Working With Relationships](#), and [Chapter 17, Working With Policies](#).

Localizing Base Types

As stated above, the Annotation, Attachment and Report types must exist in order to use these functions. However, when eMatrix is in use in a non-English language, the non-English word for these types may be used if the following variables are set in the initialization file:

MX_ANNOTATION_TYPE sets the non-English word used for Annotations. A Business Object Type of the name specified here must be defined in order for the Annotation function to operate properly when used in a non-English setting. The default is Annotation.

MX_ATTACHMENT_TYPE sets the non-English word used for Attachments. A Business Object Type of the name specified here must be defined in order for the Attachment function to operate properly when used in a non-English setting. The default is Attachment.

MX_REPORT_TYPE sets the non-English word used for Reports. A Business Object Type of the name specified here must be defined in order for the Report function to operate properly when used in a non-English setting. The default is Report.

In order to change the menu options for report, annotation and attachment to the non-English words, the matrix.txt file must be translated and imported. Refer to the *System Manager Guide* for more information.

Changes to matrix.txt are for desktop only and do not affect PowerWeb.

Deleting a Type

If a type is no longer required, you can delete it with the Delete Type statement:

```
delete type NAME;
```

NAME is the name of the type to be deleted.

When this statement is processed, eMatrix searches the list of business object types. If the name is not found, an error message is displayed. If the name is found and the type does not derive another type, and there are no business objects of this type, the type is deleted. If it is a parent type, you must first delete the types that are derived from it.

For example, to delete the type named “Federal Individual Income Tax,” enter the following MQL statement:

```
delete type "Federal Individual Income Tax";
```

eMatrix will delete the type if:

- There are no business objects of this type.
- There are no types derived from this type.

Working With Relationships

Overview of Relationships	338
Collecting Data for Defining Relationships	339
Defining a Relationship	341
Attribute Clause	341
Trigger Clause	342
Description Clause	343
Icon Clause	343
To and From Clauses	344
PreventDuplicates Clause	352
Hidden Clause	353
Property Clause	353
Copying and/or Modifying a Relationship Definition	354
Copying (Cloning) a Relationship Definition	354
Modifying a Relationship Definition	354
Connection End Modifications	356
Deleting a Relationship	358
Working with Relationship Instances	359
Modifying Relationships	359
Connection IDs	359
Modifying Attributes of a Connection	360
Freezing Connections	360
Thawing Connections	360
Deleting Connections	360
Printing Connections	360

Overview of Relationships

A *relationship* definition is used along with the policy to implement business practices. Therefore, they are relatively complex definitions, usually requiring planning.

For example, in manufacturing, a component may be contained in several different assemblies or subassemblies in varying quantities. If the component is later redesigned, the older design may then become obsolete. In eMatrix, the component objects could be connected to the various assembly and subassembly objects that contain it. Each time objects are connected with this relationship, the user could be prompted for the quantity value for the relationship instance. If the component is later redesigned, the older design may become obsolete. When a revision of the component object is then created, the relationship would disconnect from the original and connect to the newer revision. If the component is cloned because a similar component is available, the cloned component may or may not be part of the assembly the original component connects to. The connection to the original should remain but there should be no connection to the cloned component.

For the process to work in this fashion, the relationship definition would include the attribute “quantity.” The cardinality would be “many to many” since components could be connected to several assemblies and assemblies can contain many components. The revision rule would be “float” so new revisions would use the connections of the original. The clone rule would be “none” so the original connection remains but no connection is created for the clone.

You must be a Business Administrator to define relationships. (Refer also to your Business Modeler Guide.) Relationships are typically initially created through MQL when all other primary administrative objects are defined. However, if a new relationship must be added, it can be created with Business Administrator.

Collecting Data for Defining Relationships

In an MQL schema definition script, relationship definitions should be placed after attributes and types. Before writing the MQL statement for adding a relationship definition, the Business Administrator must determine:

- Of the types of business objects that have been defined, which types will be allowed to connect directly to which other types?
- What is the nature and, therefore, the name of each relationship?
- Relationships have two ends. The *from* end points to the *to* end. Which way should the arrow (in the Indented and Star Browsers) point for each relationship?
- What is the meaning of the relationship from the point of view of the business object on the *from* side?
- What is the meaning of the relationship from the point of view of the business object on the *to* side?
- What is the cardinality for the relationship at the *from* end? Should a business object be allowed to be on the *from* end of only one or many of this type of relationship?
- What is the cardinality for the relationship at the *to* end? Should a business object be allowed to be on the *to* end of only one or many of this type of relationship?
- When a business object at the *from* end of the relationship is revised or cloned, a new business object (similar to the original) is created. What should happen to this relationship when this occurs? The choices for revisions and clones are: none, replicate, and float.

Should the relationship stay on the original and not automatically be connected to the new revision or clone? If so, pick none.

Should the relationship stay on the original and automatically connect to the new revision or clone? If so, pick replicate.

Should the relationship disconnect from the original and automatically connect to the new revision or clone? If so, pick float.

- When a business object at the *to* end of the relationship is revised or cloned, a new business object (similar to the original) is created. What should happen to this relationship when this occurs? The choices are the same as for the *from* end.
- What attributes, if any, belong on the relationship? Quantity, Units, and Effectivity are examples of attributes which logically belong on a relationship between an assembly and a component rather than on the assembly or component business object. Each instance, or use of the relationship, will have its own values for these attributes which apply to the relationship between the unique business objects it connects.

Use a table like the one below to collect the information needed for relationship definitions.

Relationship Name	_____	_____	_____	_____	_____	_____
From Type	_____	_____	_____	_____	_____	_____
From Meaning	_____	_____	_____	_____	_____	_____
From Cardinality	_____	_____	_____	_____	_____	_____
From Rev Behavior	_____	_____	_____	_____	_____	_____
From Clone Behavior	_____	_____	_____	_____	_____	_____
To Type	_____	_____	_____	_____	_____	_____
To Meaning	_____	_____	_____	_____	_____	_____
To Cardinality	_____	_____	_____	_____	_____	_____
To Rev Behavior	_____	_____	_____	_____	_____	_____
To Clone Behavior	_____	_____	_____	_____	_____	_____
Attributes	_____	_____	_____	_____	_____	_____

Defining a Relationship

A relationship between two business objects is defined with the Add Relationship statement:

```
add relationship NAME [ADD_ITEM {ADD_ITEM}] ;
```

NAME is the name of the relationship you are defining. The relationship name is limited to 127 characters. For additional information, refer to *Business Object Name* in Chapter 34.

ADD_ITEM is an Add Relationship clause which provides more information about the relationship you are creating. The Add Relationship clauses are:

attribute ATTRIBUTE_NAME { , ATTRIBUTE_NAME }
trigger EVENT_TYPE TRIGGER_TYPE PROG_NAME [input ARG_STRING]
description VALUE
icon FILENAME
from ADD_SUB_ITEM { , ADD_SUB_ITEM }
to ADD_SUB_ITEM { , ADD_SUB_ITEM }
[! not] preventDuplicates
[! not] hidden
property NAME [to ADMINTYPE NAME] [value STRING]

Only the From and To clauses are required to make a relationship usable. However, each clause is beneficial when defining relationships. In the sections that follow, the clauses and the arguments they use are discussed.

Attribute Clause

The Attribute clause of the Add Relationship statement associates one or more attributes with a relationship. To assign an attribute to a relationship, it must be previously defined. Attributes are useful in providing information specific to the connection you are making. In some cases the information is included in the business objects; in other cases, it is more appropriate to associate the information with the body of the connection. When making a connection, the user will fill in the attribute values.

For example, many different Assembly objects might use a Component object. The cost for installing the component into each assembly may differ considerably. Therefore, you may want to define an attribute called “Installation Cost” and associate it with the component relationship. Whenever a connection is made between a Component object and an Assembly object, the user can insert the cost associated with that connection.

While you may originally define an attribute for use in a relationship, it can also be used in other relationship definitions or type definitions. Therefore, it is possible for the attribute to be contained within the object itself.

As another example, a Quantity attribute is assigned to an assembly object to track the number of components used. You also can assign the Quantity attribute to a relationship to track the number of times the component is required for an Assembly. Since the quantity of the component may differ from assembly to assembly, the relationship records the amount as part of its definition. When a specific Component object is connected to a

particular Assembly object, the user automatically has a means of inserting the quantity information.

An Add Relationship statement may have many or no Attribute clauses depending on the types of objects being connected and why. For example, the following statement associates three attributes with the relationship named “Assembly Relationship”:

```
add relationship "Assembly Relationship"
    description "Identifies component objects used in an assembly object"
    attribute Quantity
    attribute Units
    attribute "Installation Cost"
    from
        type Assembly
        meaning "Is composed of"
        cardinality n
        revision float
        clone none
    to
        type Component
        meaning "Is used by"
        cardinality n
        revision float
        clone none;
```

Once this statement is processed and the relationship is defined, the user can create connections between instances of Component type objects and instances of Assembly type objects by using the Connect Businessobject statement (as described in [Making Connections Between Business Objects](#) in Chapter 35). These connections have three fields where the user can define the values for the attributes: Quantity, Units, and Installation Cost. Although the user is not required to enter values for the attributes, they are always available.

Trigger Clause

Event Triggers allow the execution of a Program object to be associated with the occurrence of an event. The following Relationship events support Triggers:

- create
- delete
- modifyattribute
- freeze
- thaw

For example, when a relationship is instantiated (created), a trigger could check that an attribute value is equal to a certain value, and notification of the connection could be sent to an appropriate user. In fact, if the attribute value did not meet a specified set of criteria, a different event could replace the original. These transactions are written into program objects which are then called by the trigger.

Relationship Triggers use the following syntax:

```
trigger EVENT_TYPE TRIGGER_TYPE PROG_NAME [ input ARG_STRING ]
```

EVENT_TYPE is any of the valid events for Relationships: create, delete, modifyattribute, freeze, thaw.

TRIGGER_TYPE is Check, Override, or Action. Refer to [Types of Triggers](#) in Chapter 20.

PROG_NAME is the name of the Program object that will execute when the event occurs.

ARG_STRING is a string of arguments to be passed into the program. When you pass arguments into the program they are referenced by variables within the program. Variables 0, 1, 2... etc. are reserved by the system for passing in arguments.

Environment variable "0" always holds the program name and is set automatically by the system.

Arguments following the program name are set in environment variables "1", "2",... etc.

See [Using the Runtime Program Environment](#) in Chapter 19 for additional information on program environment variables.

For example:

```
add relationship "Assembly Relationship"
    description "Identifies the component objects used in an assembly object"
    attribute Quantity
    attribute Units
    attribute "Installation Cost"
    from
        type Assembly
    to
        type Component
    trigger create check "Quantity Check"
    trigger create override "Use Alternate Relationship"
    trigger create action "Notify John";
```

Refer to [Designing Triggers](#) in Chapter 20 for more information.

Description Clause

The Description clause of the Add Relationship statement provides general information for both you and the user about the function of the relationship. There may be subtle differences between some relationships; the Description clause enables you to point out the differences to the user.

There is no limit to the number of characters you can include in the description. However, keep in mind that the description is displayed when the mouse pointer stops over the relationship in a chooser. Although you are not required to provide a description, this information is helpful when a choice is requested.

For example, assume you have two relationships: Comment and Annotation. Which relationship should the user use to connect a documentation object to a drawing object? A Description clause for each of these two relationships should:

- Provide reasons why each relationship is defined.
- Indicate the differences between them.

For example, in these statements you can determine the relationship to use:

```
add relationship Comment
    description "Connects documentation objects to projects, plans, and specs";
add relationship Annotation
    description "Connects documentation objects to drawings and schematics";
```

Icon Clause

The Icon clause of the Add Relationship statement associates an image with a business object relationship. For example, you may want to use an icon that represents the types of

objects being linked. Icons help users locate and recognize items by associating a special image with the relationship. You can assign a special icon to the new relationship or use the default icon. The default icon is used when in view-by-icon mode. Any special icon you assign is used when in view-by-image mode. When assigning a unique icon, you must use a GIF image file. Refer to *Icon Clause* in Chapter 1 for a complete description of the Icon clause.

GIF filenames should not include the @ sign, as that is used internally by eMatrix.

To and From Clauses

The To and From clauses of the Add Relationship Statement define the ends of the connections. These clauses identify:

- The types of business objects that can have this relationship.
- The meaning of each connection end.
- The rules for maintaining the relationship.

All relationships will occur between two business objects. These objects may be of the same type or different types. When looking at a relationship, the objects are connected TO and FROM one another.

In some relationships, you can assign either object to either end. However, the To and From labels help you identify the reason for the connection.

You must define the connection with one To clause and one From clause in your Add Relationship statement. If either clause is missing or incomplete, the relationship will not be created.

Both the To and From clauses use the same set of subclauses because they define the same information about each connected business object. The separate values you use to define each connection end will vary according to the type of connection you are making and the types of objects involved.

When you define one end of a connection, only one subclause is required: type.

type TYPE_NAME { , TYPE_NAME }
type all
meaning VALUE
cardinality CARDINAL_ID
revision REVISION_RULE
clone CLONE_RULE
[! not]propagatemodify

type defines the types of business objects the user can use for each connection end within the relationship.

meaning (optional) helps the user identify the purpose of the objects being connected. The meaning is limited to 127 characters.

cardinality, revision and clone (all optional) deal with the number of relationships that the object can have on each end and how those relationships are maintained when one of the connected objects is revised or cloned. There must be a level of agreement between the clauses in order for both to work properly. Cardinality defaults to MANY, revision and clone default to NONE.

`propagatemodify` specifies how modifications to the relationship instance are reflected in the modified timestamp of the objects on each end. This is helpful for monitoring changes made on a certain day or within a specified time period. With this switch on, objects whose only modifications have been to their relationships can be found by queries searching on modification date values. The default is off, so when adding relationships that don't require this feature, the `notpropagatemodify` clause is not necessary.

These clauses are discussed in the paragraphs that follow.

Type Subclause

The Type subclause specifies the types of business objects that can be used for each connection end within the relationship. Types must already be defined in eMatrix. You must specify at least one business object type at each end in order for the relationship to be valid.

When both ends of the connection involve a single business object type, the relationship name can reflect the types being connected. For example, a relationship name of "Model and Drawing" might always refer to a connection between an electronic drawing and a physical model made from the drawing.

You can define type as a single business type or several types.

A name of "Alternative Component" might always refer to a connection between two component objects used interchangeably in an assembly. In both examples, the name reflects the type of objects connected by the relationship.

An Add Relationship statement to define the relationships between components, assemblies, and subassemblies, might appear as:

```
add relationship "Part Usage"
description "Identifies the objects used in an assembly or subassembly"
attribute Quantity
attribute "Installation Time"
from
  type Assembly, Subassembly
  meaning "Is composed of"
  cardinality n
  revision float
  clone none
to
  type Component, Subassembly
  meaning "Is used by"
  cardinality n
  revision float
  clone none;
```

When a connection end can be assigned multiple business types, the name of the relationship needs to be more generic. For example, a relationship named "Part Usage" might have one connection end that is either an Assembly or Subassembly object type. The other connection end is either a Component or Subassembly object type. While you have only one relationship, you actually have four types of connections you can make:

- Component objects and Subassembly objects
- Component objects and Assembly objects

- Subassembly objects and Subassembly objects
- Subassembly objects and Assembly objects

This enables you to relate all components and subassemblies to their larger subassemblies and assemblies without defining a relationship for each connection type.

A variation of the Type subclause uses the keyword all in place of one or more type names (type all). When this keyword is used, all business object types defined within eMatrix are allowed with the named relationship. This means that the specified relationship end can have any object type. Be aware that any additional object types added at a later date will be allowed within the relationship. This can create problems if the new object types are incompatible with the relationship ends. Therefore, you should use the type all subclause with care.

Meaning Subclause

An end's *meaning* is a descriptive phrase that identifies how the connection end relates to the other end (when viewed from the other end). The meaning helps the user identify the purpose of the objects being connected. Although the Meaning clause is not required, its use is strongly recommended.

In the example statement on the previous page, the Meaning clauses identify the arrangement when a Subassembly object is connected to another Subassembly object. The Meaning clauses identify the order.

Even when both objects are equivalent, inserting a Meaning clause is helpful. It tells the user that the order does not matter. For example, assume you wrote a relationship definition to link equivalent components. In the following definition, the Meaning clause states that both objects have the same meaning. While the user might guess the meaning from the relationship name and description, the Meaning clause eliminates doubt.

```
add relationship "Alternative Components"
  description "Identifies interchangeable components"
  from
    type Component
    meaning "Can be used in place of"
    cardinality 1
    revision none
    clone none
  to
    type Component
    meaning "Can be used in place of"
    cardinality 1
    revision none
    clone none;
```

Cardinality Subclause

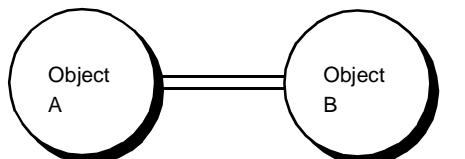
Cardinality refers to the number of relationships that can simultaneously exist between the two instances of business objects. The Cardinality subclause defines this number. When you define the cardinality, it can have one of these values:

- **One or 1**—The object can only have one connection of this relationship type at any time.
- **Many**—The object can have several relationships of this type simultaneously.

Since cardinality is defined for each end of a connection, it is possible to have three cardinal relationships between the ends:

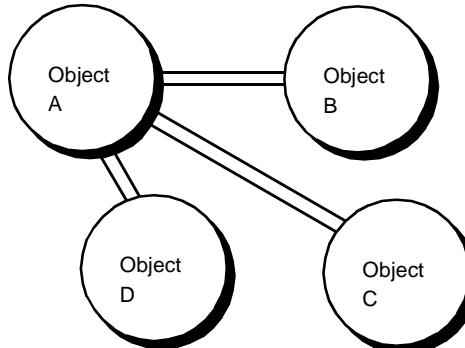
1 to 1	<i>Or:</i>	one to one
1 to n	<i>Or:</i>	n to 1
n to n	<i>Or:</i>	many to many

ONE-to-ONE



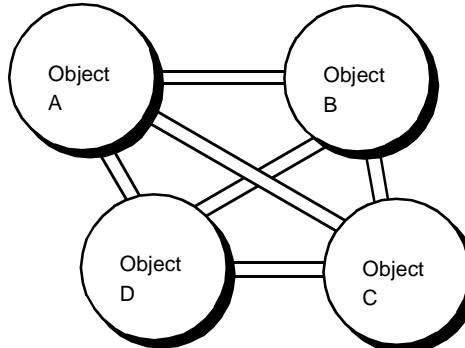
Object A can connect only with Object B.

MANY-to-ONE



Objects B, C, and D can have only one connection. Object A can have many connections.

MANY-to-MANY



All objects can have multiple connections simultaneously.

One-to-One

In a One-to-One relationship, the object on each end can be connected to only one other object with this type of relationship. An example of this type of cardinality might apply with a Change Order object connected to a Drawing object. Only one Change Order can be attached to the Drawing at any time and only one Drawing object can be attached to a Change Order.

In another example, you might have only one Customer object connected to a Flight Reservation object. The same customer cannot be on two flights simultaneously and two paying customers cannot occupy the same seat on a flight. Even a mother with a baby is contained in a single Customer object since she still represents a single paying customer.

One-to-Many Or Many-to-One

In a One-to-Many or Many-to-One relationship, the object on the “many” side can be attached to many other objects with this relationship type while the object on the “one” end cannot. An example of this type of relationship is a training course with multiple course evaluations. In an evaluation relationship, a single Training Course object can have many Course Evaluation objects attached to it. Therefore, the side of the relationship that allows the Training Course type needs a cardinality of One so that each Course Evaluation object can be connected to only one Training Course. On the other hand, the side of the relationship that allows the Course Evaluation type needs a cardinality of Many to allow many of them to use this kind of relationship to attach to the Course object.

Tip: Think of how many objects will typically exist on each end of the relationship. If it is one, the cardinality is ONE. If it is more than one, the cardinality is MANY.

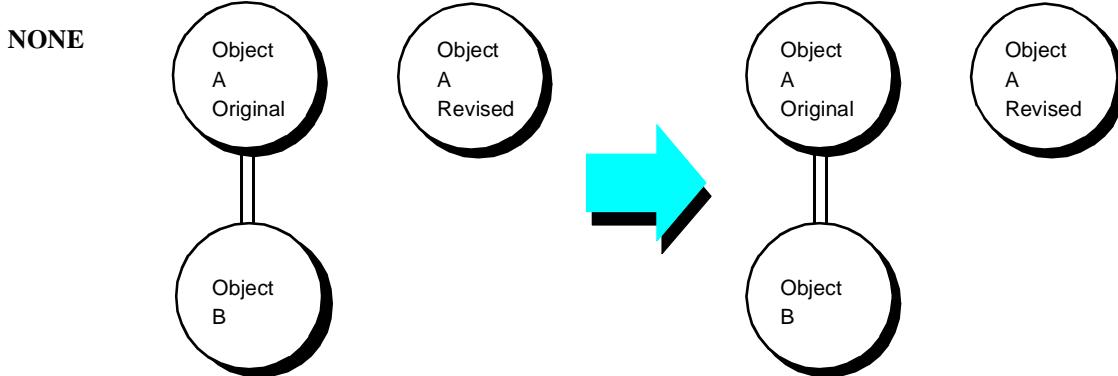
Many-to-Many

In a Many-to-Many relationship, objects on both ends of the relationship can have multiple simultaneous connections of this relationship type. This type of cardinal relationship is evident in a relationship between Component and Assembly objects. One Component object may be simultaneously connected to many different Assembly objects while one Assembly object may be simultaneously connected to many different Component objects. Both sides of the relationship are defined with a cardinality of Many since both can have more than one connection of this type at any time.

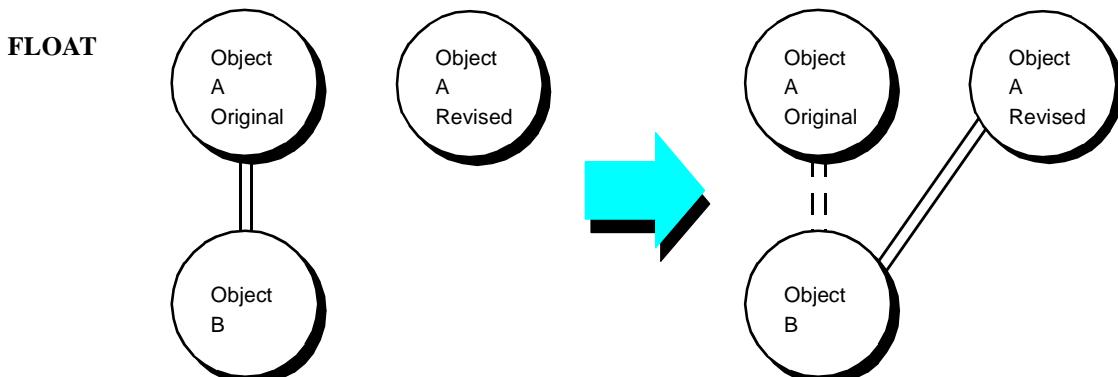
Revision Subclause

When you are defining the cardinal value for a connection end, one factor that you must consider is revision. What will happen to the relationship if one of the connection ends is revised? Will you shift the relationship to the revised object, create a second new relationship with the revised object, or simply maintain the status quo by retaining a relationship with the unrevised object? The answer is specified by the revision rule associated with each connection end, as described in the following paragraphs.

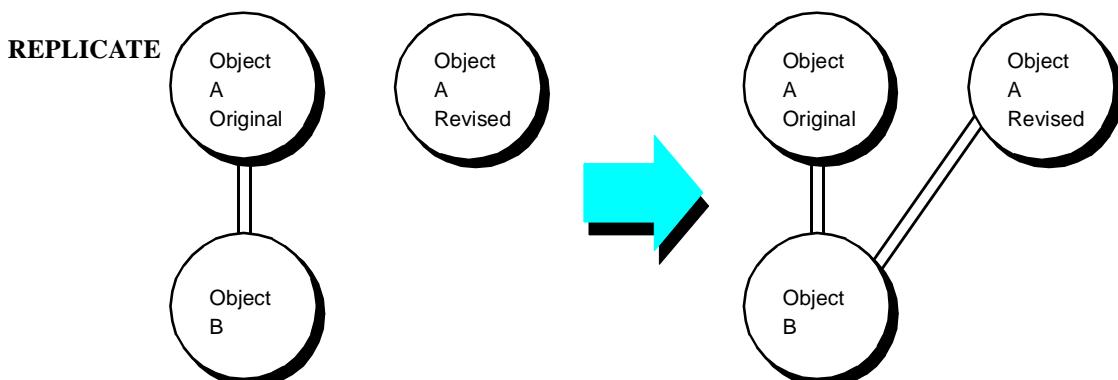
The Revision subclause identifies the rule for handling revisions of the connection object. There are three revision rules for handling revised connection ends: **None**, **Float**, and **Replicate** (as illustrated below).



Status quo is maintained. The revised object has no connection.



The connection shifts to the revised object.



A new connection is made to the revised object. The original connection is left intact.

None

When a connection end uses the None rule, nothing happens to the established connection when the end object is revised. The revised object does not automatically have a relationship attached to it after it is created. If you wanted to connect the revised object to the same object as the unrevised one, you would have to manually connect it with the Connect Business Objects statement (see *Making Connections Between Business Objects* in Chapter 35).

Using the None rule is useful when an object revision removes the need for the connection. For example, when you have a connection between a Training Course object and a Course Evaluation object, the connection may no longer be required or useful if the Training Course object is revised. While you may want to maintain the connection between the old version of the Training Course and the evaluation, the evaluation does not apply to the new version of the Training Course object. Therefore the connection end occupied by the Training Course object would use the None rule. But what of the other connection end?

If the Evaluation object is revised, it is still useful to the Training Course object. The revised object should remain connected to the Training Course object but the unrevised object no longer applies. To handle this situation, you would define the Course Evaluation end with the Float rule.

Float

The Float rule specifies that the relationship should be shifted whenever the object is revised. When the Float rule is used, the unrevised (or older version of the) object loses the connection with its other end. In its place the other end is automatically connected to the revised object. Now the older version of the object will be unattached while the newer version will have the relationship. This floating of the connection ensures that the latest versions of the object(s) are linked together.

But what if you want to maintain the old relationship while still creating a relationship with the new version? This would actually produce two connections: one between the unrevised object and its other end and one between the revised object and its other end. In this case, you would use the Replicate rule.

Replicate

The Replicate rule automatically creates new relationships whenever a connection end is revised. This results in a connection end that may have more than one simultaneous relationship of the same relationship type. For this reason, any connection end that uses the Replicate rule must also use a cardinality of “n,” as described in section *Cardinality Subclause*. If a cardinality value of 1 (one) is used, Replicate can not work.

The Replicate rule is useful when you want to keep track of former relationships. Since the old relationships are maintained while new ones are created, a user can easily see all versions that are related to a connected object. For example, you might have a relationship between a Specification object and a Specification Change object. As the Specification object is revised, you want to maintain the relationship so that you know that this Specification Change applied to this version of the Specification.

However, you also want the relationship to exist between the revised Specification and the Specification Change. This new relationship enables you to trace the history of the changes made and the reasons for them. In this situation, the Specification object should use the Replicate revision rule while the Specification Change object might use the Float or Replicate rule.

An Add Relationship statement for the Specification Change might appear as:

```
add relationship "Specification Change"
  description "Associates a change notice with a specification"
  from
    type "Specification Change Notice"
    meaning "Contains changes to be made"
    cardinality 1
    revision float
    clone replicate
  to
    type Specification
    meaning "To be changed"
    cardinality n
    revision replicate
    clone none;
```

Note that the two connection ends have different revision rules and cardinality. Remember that these values should be determined for each connection end based upon the needs and requirements for that object as it relates to the connection being made. Since you may want only the latest version of the Specification Change Notice to be attached to the Specification, Float is the best choice for the revision rule. If you wanted to keep track of all notices that were attached to the Specification, you could change the revision rule to Replicate.

Clone Subclause

Just as you must define what should happen to the relationship if one of the connection ends is revised, you must define what should happen if one of the connection ends is cloned. The same three rules available for revisions are available for clones: **None**, **Float**, and **Replicate**.

Most business rules require a clone to be treated much differently than a revision, so you may often select a different rule for clones than for revisions. For example, the None rule is often useful when a connection end is cloned. Consider a Specification Change Notice object that is connected to a Specification object. If the Specification is cloned for a new product that is very similar, it's unlikely that the Specification Change Notice applies to the cloned Specification. So the original connection between the Specification Change Notice object and Specification object should remain but no new connection is needed for the cloned Specification object.

Propagate Modify Subclause

The `propagatemodify` setting on the `from` or `to` clause controls whether or not changes to a relationship instance affect the modification timestamp of the `from/to` business object(s). When not used, changes to the relationship instance do not affect the modified date of the business objects. The `not` (or `!`) form of the subclause can be used when modifying relationships to turn the setting off, if required. The default is off, so when adding relationships that don't require this feature, the `notpropagatemodify` clause is not necessary.

For example:

```
add relationship BOM
    attribute Quantity
    to
        type Component
    from
        type Assembly
    propagatemodify;
```

Using the relationship above, changes in the quantity of the component in the assembly will be reflected in the modification date of the Assembly.

Propagate Connection Subclause

The `propagateconnection` setting on the `from` or `to` clause controls whether modification timestamps on connected business objects are recorded when the objects connect and disconnect. You can turn off modification timestamps on a per relationship basis and control the “from” and “to” end of the relationship.

This is particularly useful for “one to many” relationships where the “many” side potentially may have hundreds of objects, connected by multiple users. Such frequent timestamp logging can slow down performance and cause potential concurrency issues. Also, if you query the database for objects that have been modified, you might not want to include every business object that has merely participated in a connect/disconnect operation without having their actual definition changed.

The system will run faster and the chance of deadlocks is greatly reduced when both modification timestamps and history are turned off during connect/disconnect operations, since only the relationship tables (not the business object tables) are affected. (See *The system history setting should not be issued within program objects, since it affects all users. In these cases, the temporary MQL history off command should be used instead..*)

By default, the system updates modification timestamps on business objects every time they participate in a connect/disconnect operation. Use the `not (!)` form of the subclause to turn off these updates.

For example, there might be a Specification relationship defined for Feature objects on the “from” end and Document objects on the “to” end. You might turn off the “to” end, but leave the timestamp active on the “from” end. Whenever a connect/disconnect event is performed between these two objects, the Feature object’s timestamp is updated, but the Document object’s is not.

```
add relationship Specification
    to type Document !propagateconnection
    from type Feature;
```

PreventDuplicates Clause

A flag can be set in the relationship definition that will prevent duplicates of the relationship type to exist between the same two objects. The default is that duplicates are allowed.

For example, to prevent duplicates of the relationship Documents, use the following command:

```
add relationship Documents
    preventduplicates
```

`!preventDuplicates` would turn this feature off.

The `preventDuplicates` flag will NOT prevent a second relationship between two objects if it points in the opposite direction. For example, given `BusObjA` connected ONCE to `BusObjB` with `preventDuplicates`, connecting `BusObjA` with `preventDuplicates` to `BusObjB` will fail. Connecting `BusObjB` with `preventDuplicates` to `BusObjA` will succeed.

Hidden Clause

You can specify that the new relationship is “hidden” so that it does not appear in the Relationship chooser in eMatrix. You may want to use the hidden option if, for example, an object is under development or if it is intended only for your personal use. Hidden objects are accessible through MQL.

Property Clause

Integrators can assign ad hoc attributes, called Properties, to the relationship. Properties allow associations to exist between administrative definitions that aren’t already associated. The property information can include a name, an arbitrary string value, and a reference to another administration object. The property name is always required. The value string and object reference are both optional. The property name can be reused for different object references, that is, the name joined with the object reference must be unique for any object that has properties.

```
add relationship NAME  
    property NAME [to ADMINTYPE NAME] [value STRING];
```

For additional information on properties, see *Overview of Administration Properties* in Chapter 24.

Copying and/or Modifying a Relationship Definition

Copying (Cloning) a Relationship Definition

After a relationship is defined, you can clone the definition with the Copy Relationship statement. This statement lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy relationship SRC_NAME DST_NAME [MOD_ITEM {MOD_ITEM}];
```

SRC_NAME is the name of the relationship definition (source) to copied.

DST_NAME is the name of the new definition (destination).

MOD_ITEMS are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

Modifying a Relationship Definition

After a business object relationship is defined, you can change the definition with the Modify Relationship statement. This statement lets you add or remove defining clauses and change the value of clause arguments.

```
modify relationship NAME [MOD_ITEM {MOD_ITEM}];
```

NAME is the name of the relationship you want to modify.

MOD_ITEM is the type of modification you want to make. Each is specified in a Modify Relationship clause, as listed in the following table. Note that you only need to specify fields to be modified.

Modify Relationship Clause	Specifies that...
name NAME	The current relationship name is changed to that of the new name.
add attribute NAME	The attribute listed here is associated with this relationship.
remove attribute NAME	The attribute listed here is removed from the relationship.
description VALUE	The current description, if any, is changed to the value entered.
add trigger EVENT_TYPE TRIGGER_TYPE PROGNAME	The specified trigger is added or modified for the listed event.
remove trigger EVENT_TYPE TRIGGER_TYPE	The specified trigger type is removed from the listed event.
icon FILENAME	The image is changed to the new image in the file specified.

Modify Relationship Clause	Specifies that...
from MOD_SUB_ITEM { ,MOD_SUB_ITEM}	A modification to the <i>from</i> connection end is made. This modification may involve altering the types of objects used by the relationship and/or how the relationship should be maintained if there is a revision to an object instance. Refer to the description of <i>Connection End Modifications</i> below.
to MOD_SUB_ITEM { ,MOD_SUB_ITEM}	A modification to the <i>to</i> connection end is made. This modification may involve altering the types of objects used by the relationship and/or how the relationship should be maintained if there is a revision to an object instance. Refer to the description of <i>Connection End Modifications</i> below.
add rule NAME	The specified access rule is added.
remove rule NAME	The specified access rule is removed
preventDuplicates	The preventDuplicates flag is changed so that duplicate relationship types are not allowed.
notpreventDuplicates	The preventDuplicates flag is changed so that duplicate relationship types are allowed.
hidden	The hidden option is changed to specify that the object is hidden.
nohidden	The hidden option is changed to specify that the object is not hidden.
property NAME [to ADMINTYPE NAME] [value STRING]	The named property is modified.
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.

As you can see, each modification clause is related to the clauses and arguments that define the relationship. When you modify a business object relationship, you first name the relationship to be changed and then list the modifications. For example, to change the name of the Alternative relationship and add an attribute and a trigger, you might write this statement:

```
modify relationship Alternative
  name "Component Alternative"
  add attribute "Cost Comparison"
  add trigger create check "Cost Check"
  add trigger create override "Use Alternate Relationship"
  add trigger create action "Notify John";
```

Connection End Modifications

When you are modifying the connection ends of the relationship, the MOD_SUB_ITEM clause is similar to the clauses used to define the end objects:

Connection End Modification Clause	Specifies that...
add type all	All defined eMatrix types are allowed to be associated with this relationship end.
add type TYPE_NAME { ,TYPE_NAME }	The object type(s) listed here are allowed to be associated with this relationship end.
remove type all	All object types are removed from this relationship end.
remove type TYPE_NAME { ,TYPE_NAME }	The object type(s) listed here are removed from this relationship.
meaning VALUE	The current meaning, if any, changes to the value entered.
cardinality CARDINAL_ID	The value for cardinality is set to the value entered.
revision REVISION_RULE	The revision rule changes to the value entered.
clone CLONE_RULE	The clone rule changes to the value entered.
propagatemodify	The modification timestamp of the object on the specified end will be updated when the relationship's attribute values are modified.
[! not]propagatemodify	The modification timestamp of the object on the specified end will not be updated when the relationship's attribute values are modified.
propagateconnection	The modification timestamp of the object on the specified end will be updated during connect/ disconnect operations.
[! not]propagateconnection	The modification timestamp of the object on the specified end will not be updated during connect/ disconnect operations.

These clauses are used within the To and From clauses as they are used in the Add Relationship statement. Assume you have the following definition:

```
add relationship Comment
  description "Associates comment with related object"
  from
    type Comment
    meaning "Provides comment about"
    cardinality n
    revision none
    clone none
    propagatemodify
    !propagateconnection
  to
    type Assembly, Document, Specification, Layout
    meaning "Is commented in"
    cardinality n
    revision none
    clone none
    propagatemodify
    !propagateconnection;
```

To this definition, you might want to add a new object type that can connect to Comment objects, and you want objects on the from end to propagate modifications of the relationship to the business object. You also want to save all revised comments and include them in all revisions with the commented object. To make these changes, you might write the following Modify Relationship statement:

```
modify relationship Comment
  to
    revision replicate
  from
    add type "Process Plan" propagatemodify;
```

When this statement is processed, the definition for the Comment relationship appears as:

```
relationship Comment
  description "Associates comment with related object"
  from
    type Comment
    meaning Provides comment about
    cardinality n
    revision replicate
    clone none
    propagatemodify
  to
    type Assembly, Document, Specification, Layout,
    Process Plan
    meaning Is commented in
    cardinality n
    revision none
    clone none;
```

Deleting a Relationship

If a relationship is no longer desired between business objects, you can delete it with the Delete Relationship statement:

```
delete relationship NAME;
```

NAME is the name of the relationship to be deleted.

When this statement is processed, eMatrix searches the list of existing business object relationships. If the name is not found, an error message is displayed. If the name is found, the relationship is deleted along with all information about that relationship.

For example, to delete the relationship named “Maintenance Relationship,” enter the following MQL statement:

```
delete relationship "Maintenance Relationship";
```

After this statement is processed, the relationship is deleted and you receive an MQL prompt for another statement.

Working with Relationship Instances

Once you have defined Relationship types, as described in the sections above, connections can be made between specific business objects. Refer to [Making Connections Between Business Objects](#) in Chapter 35, for the MQL commands for connecting objects. These relationship instances can be accessed by MQL in two ways:

- by specifying the business objects on each end

Or:

- by specifying its connection ID.

The first method will work only if exactly one of a particular relationship type exists between the two listed objects. Therefore, if relationships are to be programmatically modified, the second method, using connection IDs, is the safer approach.

Modifying Relationships

Relationship instances can be accessed so that operations, such as modifying attributes, can be performed. A relationship instance can be specified by its defined name together with the objects on both connection ends as follows:

```
modify connection businessobject OBJECTID to|from businessobject OBJECTID  
relationship REL_TYPE ATTR_NAME ATTR_VAL, [ATTR_NAME ATTR_VAL];
```

OBJECTID is the OID or Type Name Revision of the business object.

REL_TYPE is the Relationship name.

ATTR_NAME is the name of the attribute.

ATTR_VAL is the new value for the attribute.

For example:

```
modify connection bus Assembly 50463 A to Component  
33457-4G relationship9 "As Designed" Quantity 5;
```

If the object is connected to another object by more than one instance of the specified relationship type, the above command generates an error message. If this message occurs, in order to make modifications, the relationship ID must be specified.

Connection IDs

To obtain the relationship IDs of all connections, select the relationship ID item in an `expand businessobject` `select` statement; for example:

```
expand bus Assembly 50402 B select relationship id;
```

The following is the result that could be saved to a file if an output clause is used in the above-referenced command:

```
1 Drawing from Drawing 50402 A  
id = 19.24.5.16  
1 Process Plan from Process Plan 50402-1 C  
id = 19.26.6.6
```

Refer to [Displaying and Searching Business Object Connections](#) in Chapter 35 for more information on the `expand` statement.

Modifying Attributes of a Connection

The returned list of IDs can be used to make attribute changes to any of the listed connections, using the following syntax:

```
modify connection ID ATTR_NAME ATTR_VAL [ATTR_NAME ATTR_VAL];
```

ID is the id of the connections to be modified.

ATTR_NAME is the attribute to be modified.

ATTR_VAL is the value to be entered into the attribute.

Attributes of relationships can be modified if the user has modify access on the objects on both ends of the relationship.

Freezing Connections

Connections can be frozen (locked) to prevent configurations from being modified. Relationships that are frozen cannot be disconnected, and their attributes cannot be modified. A user would only be allowed to freeze a relationship if they have freeze access on the objects on both ends.

```
freeze connection ID;
```

When objects are cloned or revised, their existing relationships are either floated, replicated, or removed. When a frozen relationship is floated, the new relationship instance is frozen as well. Replicated relationships take on the thawed state.

Thawing Connections

A frozen relationship can be thawed (unlocked) by using the following syntax:

```
thaw connection ID;
```

Users must have thaw access on the objects on both the to and from end of the relationship.

Deleting Connections

Connections can be deleted using the disconnect command with the connection ID:

```
disconnect connection ID;
```

Users must have todisconnect access on the to object and fromdisconnect access on the from object.

Printing Connections

Users can print the description of a connection using the print connection command. For example

```
print connection 19.24.5.16;
```

The results of the above may be something like:

```
Relationship Drawing
  From bus obj 50402
  To bus obj 50402
  relationship[Drawing].isfrozen = FALSE
  relationship[Drawing].propagatemodifyfrom = FALSE
  relationship[Drawing].propagatemodifyto = FALSE
  history
  create - user: adami time: Thu May 22, 1997 6:29:20 PM
  from: Drawing 50402 A to: Assembly 50402 B
```

If the connection ID is unknown, the following may also be used:

```
print connection from OBJECTID to OBJECTID relationship RELTYPE;
```

OBJECTID is the OID or Type Name Revision of the business object. It may also include the in VAULTNAME clause, to narrow down the search.

Note that the result of print connection with object ends specified will be ambiguous when multiple relationships between the two objects exist.

It is also possible to list only certain items to be printed about relationships. To obtain the selectable list for relationship instances, use:

```
print connection selectable;
```

Use of print relationship selectable will fail. The selectable items for connections are shown below:

```
print connection selectable;
connection selectable fields:
  name
  type.*
  attribute[ ].*
  businessobject.*
  to. *
  from. *
  history[ ].*
  isfrozen
  propagatemodifyto
  propagatemodifyfrom
  id
```

Refer to [Viewing Business Object Definitions](#) in Chapter 34 for more information about the use of selectables. Also refer to the *eMatrix Navigator Guide Appendix A Select Expressions*.

15

Working With Formats

Overview of Formats	364
Defining a Format	365
Description Clause.....	365
Creator and Type Clauses.....	366
View, Edit, and Print Clauses	366
Icon Clause.....	367
Suffix Clause.....	367
Mime Clause.....	368
Version Clause	368
Hidden Clause	368
Property Clause	369
Copying and/or Modifying a Format Definition.....	370
Copying (Cloning) a Format Definition.....	370
Modifying Format Definitions	370
Deleting a Format	372

Overview of Formats

A *format* definition in eMatrix is used to capture information about different application file formats. A format stores the name of the application, the product version, and the suffix (extension) used to identify files. It also contains the commands necessary to automatically launch the application and load the relevant files from eMatrix. Formats are the definitions used to link eMatrix to the other applications in the users' environment.

Applications typically change their internal file format from time to time. Eventually older file formats are no longer readable by the current version of the software. It is wise to create new format definitions (with appropriate names) as the applications change so that you can later find the files that are in the old format and bring them up to date.

A business object can have many file formats and they are linked to the appropriate type definition by the policy definition (see *Working With Policies* in Chapter 17).

You must be a Business Administrator to add or modify formats. (Refer also to your Business Modeler Guide.)

Defining a Format

Format definitions are created using the MQL Add Format statement. This statement has the syntax:

```
add format NAME [ADD_ITEM {ADD_ITEM}]
```

NAME is the name of the format you are creating. The format name is limited to 127 characters. For additional information, refer to *Business Object Name* in Chapter 34.

ADD_ITEM is an Add Format clause which provides more information about the format. They also provide information on how a file with that format should be processed. The Add Format clauses are:

description VALUE
creator NAME
type NAME
edit PROGRAM_OBJECT_NAME
icon FILENAME
print PROGRAM_OBJECT_NAME
suffix VALUE
mime VALUE
version VALUE
view PROGRAM_OBJECT_NAME
[! not]hidden
property NAME [to ADMINTYPE NAME] [value STRING]

Each clause and the arguments they use are discussed in the sections that follow.

Description Clause

The Description clause of the Add Format statement provides general information for both you and the user about the types of files associated with this format and the overall function of the format. There may be subtle differences between formats; the Description clause enables you to point out these differences to the user.

There is no limit to the number of characters you can include in the description. However, keep in mind that the description is displayed when the mouse pointer stops over the format in a chooser. Although you are not required to provide a description, this information is helpful when a choice is requested.

When a user has a file to check in, the description guides the user as to which format will properly process the file. If the file is assigned the wrong format, the user may be unable to fully access or process the file once it is in the database.

For example, assume you have two types of word processing programs commonly used to create text documents. You want to distinguish between the file formats created by the programs. So, you create two formats: one for each word processing program. Each

format will define the word processing program used to view, edit, and print a file that was originally created with that program.

When you create the formats, you should assign names that have meaning to both you and the user. In this example, you could use the names of the programs: TextTYPE and BestBooks. Then you can provide more information by including a Description clause in each definition:

```
add format "TextTYPE"
    description "Use for text documents created with TextTYPE";
add format "BestBooks"
    description "Use for text documents created with BestBooks";
```

Creator and Type Clauses

The Creator and Type fields are Macintosh file system attributes (like Protection and Owner on UNIX systems). They should not be confused with eMatrix users or types. The following is an example Creator clause of the Add Format statement:

```
creator 'MPSX'
```

The following is an example Type clause of the Add Format statement:

```
type 'TEXT'
```

This would identify a script file created by the Macintosh toolserver. Both fields are four bytes in length and are generally readable ASCII. If you specify a value for only one of the two clauses, the other clause assumes the same value. The values for creator and type are registered with Apple for each Macintosh application. When a file is checked out to a Macintosh, these attribute settings will be applied. If Macintoshes are not used, the fields can be left blank.

View, Edit, and Print Clauses

The View, Edit, and Print clauses specify the program to use to view (open for view), edit (open for edit), or print files checked into the format. When you specify the program, you are actually specifying the name of the program object that represents the program.

For Windows platforms, if you want to open files for view, edit, or print based on their file extensions and definitions in the Windows Registry, you can leave out the corresponding clause. For example, by default Windows uses MS Paint to open files with a file extension of .bmp. Keep in mind that each user's PC contains its own Windows Registry database, which is editable; the databases are not shared between computers. If you want to provide a more complex and flexible format that will use the file association mechanism of windows, refer to [Format Definition Example Program](#) in Chapter 19.

Program object requirements

To be used in a format definition, a program object definition must include these characteristics:

- The needsbusinessobject clause must be true.
- The code clause must contain the command needed to execute the program and the syntax for the command must be appropriate for the operating system.
- The code clause should end with the \$FILENAME macro so the program opens any file. Enclose the macro in quotes to ensure that files with spaces in their names are opened correctly.

For more information on defining program objects for use in a format definition, see [Code Clause](#) in Chapter 19.

Syntax

The View, Edit, and Print clauses of the Add Format statement use this syntax:

```
view PROGRAM_OBJECT_NAME  
edit PROGRAM_OBJECT_NAME  
print PROGRAM_OBJECT_NAME
```

For example, the following is a sample format definition for CADplus, a computer aided design system:

```
add format CADplus  
description "CADplus Computer Aided Design System"  
version 10  
suffix ".cad"  
view CADview.exe  
edit CADedit.exe;
```

After this format is defined, eMatrix can open a file checked in with this format using CADview for viewing or using CADedit for editing.

Icon Clause

The Icon clause of the Add Format statement associates a special image with the format. If you defining a format for text files, you might associate an icon that shows a text page to distinguish word processing formats from image files, mail files, report files, and so on. Icons help users locate and recognize items by associating a special image with the format. The default icon is used when in view-by-icon mode. Any special icon you assign is used when in view-by-image mode. When assigning a unique icon, you must use a GIF image file. Refer to [Icon Clause](#) in Chapter 1 for a complete description of the Icon clause.

GIF filenames should not include the @ sign, as that is used internally by eMatrix.

Suffix Clause

The Suffix clause of the Add Format statement specifies the default suffix for the format. If an object is selected that contains no files, “open for edit” generates the name of the file from the object name. eMatrix attempts to open a file with that name and the default format suffix.

Assume you want to add a note to a business object. You might use the TextTYPE or BestBooks word processing programs to create the note. TextTYPE uses a default file suffix of .text for document files and BestBooks uses .bb. These suffixes enable users to quickly identify file types. For example:

```
add format "TextTYPE"  
description "For documents created with TextTYPE"  
version 3.1  
suffix ".tex";  
add format "BestBooks"  
description "For documents created with BestBooks"  
version 6.0  
suffix ".bb";
```

After these definitions are made, any file that uses a TextTYPE format will have a suffix of `.tex` and any file that uses a BestBooks format will have a suffix of `.bb`.

The suffix specified in the Format is not used in the launching mechanism—the file itself is passed to the operating system and its extension (or suffix) is used to determine what application should be opened.

Mime Clause

You can specify the MIME (Multi-Purpose Internet Mail Extension) type for a format. MIME types are used when files are accessed via a Web browser. To specify a MIME type, use the mime clause in the format definition:

```
creator VALUE
```

VALUE is the content type of the file. The format of VALUE is a type and subtype separated by a slash. For example, `text/plain` or `text/jsp`.

The major MIME types are application, audio, image, text, and video. There are a variety of formats that use the application type. For example, `application/x-pdf` refers to Adobe Acrobat Portable Document Format files. For information on specific MIME types (which are more appropriately called “media” types) refer the Internet Assigned Numbers Authority Website at <http://www.ietf.org/rfc/rfc2046.txt>. The IANA is the repository for assigned IP addresses, domain names, protocol numbers, and has also become the registry for a number of Web-related resources including media types.

To find the MIME types defined for a particular format, use the following command:

```
print format FORMAT_NAME select mime;
```

Version Clause

The Version clause of the Add Format clause identifies the version number of the software required to process the file. The software version is useful when tracking files created under different software releases. Upward and downward compatibility is not always assured between releases. If you install a new software release that cannot process existing files, you can create a new format for the new release and leave the old format in place. The old format automatically references the older version of the software while the new format references the new version.

eMatrix does not check the version number against the software you are using. You can enter any value. However, you should use the actual version number or identifier if possible. For example:

```
add format ASCII version Standard;  
add format "TextTYPE" version 3.1;
```

Hidden Clause

You can specify that the new format is “hidden” so that it does not appear in the checkin/checkout list of formats in eMatrix. You may want to use the hidden option if, for example, an object is under development or if it is intended only for your personal use. Hidden objects are accessible through MQL.

Property Clause

Integrators can assign ad hoc attributes, called Properties, to the format. Properties allow associations to exist between administrative definitions that aren't already associated. The property information can include a name, an arbitrary string value, and a reference to another administration object. The property name is always required. The value string and object reference are both optional. The property name can be reused for different object references, that is, the name joined with the object reference must be unique for any object that has properties.

```
add format NAME  
    property NAME [to ADMINTYPE NAME] [value STRING];
```

For additional information on properties, see *Overview of Administration Properties* in Chapter 24.

Copying and/or Modifying a Format Definition

Copying (Cloning) a Format Definition

After a format is defined, you can clone the definition with the Copy Format statement. This statement lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy format SRC_NAME DST_NAME [MOD_ITEM] {MOD_ITEM};
```

SRC_NAME is the name of the format definition (source) to copied.

DST_NAME is the name of the new definition (destination).

MOD_ITEMS are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

Modifying Format Definitions

After a format is defined, you can change the definition with the Modify Format statement. This statement lets you add or remove defining clauses and change the value of clause arguments:

```
modify format NAME [MOD_ITEM] {MOD_ITEM};
```

NAME is the name of the format you want to modify.

MOD_ITEM is the type of modification you want to make.

There are different types of modifications you can make. Each modification is specified in a Modify Format clause, as listed in the following table. Note that you only need to specify fields to be modified.

Modify Format Clause	Specifies that...
description VALUE	The current description, if any, is changed to the value entered.
creator NAME	For Macintosh only, the current creator name is changed to the value entered.
edit PROGRAM_OBJECT_NAME	The program object that represents the program to use to open the business object file for editing or modification.
icon FILENAME	The image is changed to the new image in the file specified.
name NEW_NAME	The current format name is changed to that of the new name entered.
print PROGRAM_OBJECT_NAME	The program object that represents the program to use to print the business object file.
suffix VALUE	The default file suffix specified is used when creating new files.
type NAME	For Macintosh only, the current type name is changed to the value entered.
mime VALUE	The MIME type for the format, which is used when a file is accessed via a Web browser.

Modify Format Clause	Specifies that...
version VALUE	The version number is set for the software processing a file with this format.
view PROGRAM_OBJECT_NAME	The program object that represents the program to use to open the business object file for viewing.
hidden	The hidden option is changed to specify that the object is hidden.
nohidden	The hidden option is changed to specify that the object is not hidden.
property NAME [to ADMINTYPE NAME] [value STRING]	The named property is modified.
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.

As you can see, each modification clause is related to the clauses and arguments that define the format. For example, the following statement changes the name and version of the format named “TextTYPE Version 9.1”:

```
modify format "TextTYPE Version 9.1"
    name "TextTYPE Version 10"
    version 10.0;
```

When modifying a format, remember the question of upward and downward compatibility between software versions. Since all files with the defined format are effected by the change, you should test sample files or read the release notes to determine whether or not old files will be negatively effected. If they will be, you may want to create a new format for the new software version rather than modify the existing format definition.

In some cases, the suffix will be different for documents created in a new release of the application software. Therefore, a separate format is required (at least until all files are updated).

Deleting a Format

If a format is no longer required, you can delete it with the Delete Format statement:

```
delete format NAME;
```

NAME is the name of the format to be deleted.

When this statement is processed, eMatrix searches the list of formats. If the name is not found, an error message is displayed. If the name is found and there are no files with that format in the database, the format is deleted. If there are files that use that format within the database, they must be reassigned or deleted from the business object before you can remove the format from the format list.

For example, delete the TextTYPE Version 9.1 format, enter the following MQL statement:

```
delete format "TextTYPE Version 9.1";
```

After this statement is processed, the format is deleted and you receive an MQL prompt for another statement.

16

Working With Rules

Rule Defined.....	374
Creating Rules	375
Creating a Rule.....	375
Description Clause.....	375
Icon Clause.....	375
Hidden Clause	376
Property Clause.....	376
Assigning Access.....	376
Copying and/or Modifying a Rule Definition	379
Copying (Cloning) a Rule Definition.....	379
Modifying a Rule Definition	379
Assigning Rules.....	381
Deleting a Rule.....	382

Rule Defined

Use rules to limit user access to attributes, forms, programs, and relationships. Unlike policies, rules control access to these administrative objects regardless of the object type or state. For example, a policy might allow all users in the Engineering group to modify the properties of Design Specification objects when the objects are in the Planning state. But you could create a rule to prevent those users from changing a particular attribute of Design Specifications, such as the Due Date. In such a case, Engineering group users would be unable to modify the Due Date attribute, no matter what type of object the attribute is attached to. For a explanation of how rules work with other objects that control access, see [Which Access Takes Precedence Over the Other?](#) in Chapter 10.

When you create a rule, you define access using the three general categories used for assigning access in policies: public, owner, and user (specific person, group, role, or association). For a description of these categories, see [Policies](#) in Chapter 10.

Creating Rules

Creating a Rule

Rules are defined using the Add Rule statement:

```
add rule NAME [ADD_ITEM {ADD_ITEM}];
```

NAME is the name of the rule you are creating. All rules must have a name assigned. When you create a rule, you should assign a name that has meaning to both you and the user. The rule name is limited to 127 characters. For additional information, refer to *Business Object Name* in Chapter 34.

ADD_ITEM is an Add Rule clause which defines information about the rule including the description, icon, and access masks. The Add Rule clauses are:

description VALUE
icon FILENAME
[! not]hidden
property NAME [to ADMINTYPE NAME] [value STRING]
owner ACCESS {,ACCESS}
public ACCESS {,ACCESS}
user USER_NAME ACCESS {,ACCESS} [filter EXPRESSION]

Some of the clauses are required and some are optional, as described in the sections that follow.

Description Clause

The Description clause of the Add Rule statement provides general information about the types of accesses associated with the rule. Since there may be subtle differences between rules, the Description clause enables you to point out these differences.

There is no limit to the number of characters you can include in the description. However, keep in mind that the description is displayed when the mouse pointer stops over the rule in a chooser. Although you are not required to provide a description, this information is helpful when a choice is requested.

Icon Clause

The Icon clause of the Add Rule statement associates an image with a rule. You can assign a special icon to the new rule or use the default icon. The default icon is used when in view-by-icon mode. Any special icon you assign is used when in view-by-image mode. When assigning a unique icon, you must use a GIF image file. Refer to *Icon Clause* in Chapter 1 for a complete description of the Icon clause.

GIF filenames should not include the @ sign, as that is used internally by eMatrix.

Hidden Clause

You can specify that the new rule is “hidden” so that it does not appear in the chooser in eMatrix. Users who are aware of the hidden rule’s existence can enter its name manually where appropriate. Hidden objects are accessible through MQL.

Property Clause

Integrators can assign ad hoc attributes, called Properties, to the rule. Properties allow associations to exist between administrative definitions that aren’t already associated. The property information can include a name, an arbitrary string value, and a reference to another administration object. The property name is always required. The value string and object reference are both optional. The property name can be reused for different object references, that is, the name joined with the object reference must be unique for any object that has properties.

```
add rule NAME  
    property NAME [to ADMINTYPE NAME] [value STRING];
```

For additional information on properties, see *Overview of Administration Properties* in Chapter 24.

Assigning Access

Each access form is used to control some aspect of a business object’s use. With each access, other than `all` and `none`, you can either *grant* the access or explicitly *deny* the access. You deny an access by entering `not` (or `!`) in front of the access in the statement. For every administrative object (attribute, form, program, relationship) governed by a rule, a user has access only if the rule specifically grants the user access. If the user isn’t granted access by the rule, the user won’t have access, even if the policy grants access to the user.

For example, suppose you don’t want anyone to modify an attribute called `Priority` unless they belong to the Management role. However, everyone should be able to view the attribute’s values. (Assume the person and policy definitions grant the appropriate accesses.) You would create a rule that governs the `Priority` attribute and define the following accesses:

Owner: read
Public: read
Management role: all

The available accesses are summarized in [Accesses](#) in Chapter 10.

It is important to keep in mind that while the complete list of access items is available when creating rules, when they are actually used, only the applicable privileges are checked. The table below shows the accesses each administrative type uses:

Accesses Used By:				
Attributes	Forms	Programs	Relationships	
read	viewform	execute	toconnect fromconnect	freeze
modify	modifyform		todisconnect fromdisconnect	thaw
Owner Access does not apply to Relationships.				

Accesses Used By:				
Attributes	Forms	Programs	Relationships	
			changetype	modify (attributes)
Owner Access does not apply to Relationships.				

Owner Clause

The Owner Clause of the Add Rule statement specifies the maximum amount of access the owner of a business object will be allowed. When this clause is used, you can select from many different forms of access. For example when creating a relationship rule you might use:

```
add rule DocumentsRelRule owner toconnect, fromconnect,
      todisconnect, fromdisconnect, changetype;
```

Public Clause

The Public Clause of the Add Rule statement specifies the maximum amount of access that all eMatrix users will be allowed. When this clause is used, you can select from many different forms of access. For example when creating a form rule you might use:

```
add rule RequisitionForm owner viewform, modifyform public
      viewform;
```

User Clause

The User Clause of the Add Rule statement specifies the maximum amount of access allowed for the specified USER_NAME, which can be any person, group, role, or association already defined to eMatrix. When this clause is used, you can select from many different forms of access. For example when creating an attribute rule you might use:

```
add rule CostAttribute
      owner Read
      user Finance read, modify
      public none;
```

User access lists defined on a rule can accept a filter expression in order to grant or deny access to a specific user. If the filter expression evaluates to “true,” the specified access will be granted; otherwise the access is denied. This provides an additional level of access granularity, which increases security and reduces overall management problems by reducing the number of policies or rules required.

Expression access filters can be any expression that is valid in the eMatrix system. Expressions are supported in various modules of the eMatrix system, for example, query where clauses, expand, filters defined on workspace objects such as cues, tips and filters,

etc. See [Working With Expression Access Filters](#) in Chapter 10 for details on how access filters can be used.

```
add rule ExportAllowed
owner read
public none
user 'Hydraulic Products, Inc.' read, modify checkin checkout
filter context.user.property[Export Allowed].value;
```

In this case, the selected access (read, modify, checkout, checkin) would be allowed only:

- if context user belongs to the group “Hydraulic Products, Inc.” and
- if `context.user.property[Export Allowed].value` is evaluated as true.

Since you must have at least read access to the business object in order to evaluate the filter, normal access checks must be disabled while an access filter is being evaluated.

Copying and/or Modifying a Rule Definition

Copying (Cloning) a Rule Definition

After a rule is defined, you can clone the definition with the Copy Rule statement. This statement lets you duplicate rule definitions with the option to change the value of clause arguments:

```
copy rule SRC_NAME DST_NAME [MOD_ITEM] {MOD_ITEM};
```

SRC_NAME is the name of the rule definition (source) to be copied.

DST_NAME is the name of the new definition (destination).

MOD_ITEMS are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

Modifying a Rule Definition

After a rule is defined, you can change the definition with the Modify Rule statement. This statement lets you add or remove defining clauses and change the value of clause arguments:

```
modify rule NAME [MOD_ITEM {MOD_ITEM}];
```

NAME is the name of the rule to modify.

MOD_ITEM is the type of modification to make. Each is specified in a Modify Rule clause, as listed in the following table. Note that you only need to specify fields to be modified.

Modify Rule Clause	Specifies that...
name NAME	The current name is changed to the new name.
description VALUE	The current description, if any, is changed to the value entered.
icon FILENAME	The image is changed to the new image in the file specified.
hidden	The hidden option is changed to specify that the object is hidden.
nothidden	The hidden option is changed to specify that the object is not hidden.
add owner ACCESS {,ACCESS}	The specified owner access is added. Values for ACCESS can be found in the <i>Accesses</i> in Chapter 10.
remove owner ACCESS {,ACCESS}	The specified owner access is removed. Values for ACCESS can be found in the <i>Accesses</i> in Chapter 10.
add public ACCESS {,ACCESS}	The specified public access is added. Values for ACCESS can be found in the <i>Accesses</i> in Chapter 10.
remove public ACCESS {,ACCESS}	The specified public access is removed. Values for ACCESS can be found in the <i>Accesses</i> in Chapter 10.

Modify Rule Clause	Specifies that...
add user USER_NAME ACCESS {,ACCESS} [filter EXPRESSION]	The specified user access is added. USER_NAME defines the person, group, role or association for which the rule is being modified. Values for ACCESS can be found in the Accesses in Chapter 10.
remove user USER_NAME ACCESS {,ACCESS} [filter EXPRESSION]	The specified user access is removed. USER_NAME defines the person, group, role or association for which the rule is being modified. Values for ACCESS can be found in the Accesses in Chapter 10.
property NAME [to ADMINTYPE NAME] [value STRING]	The named property is modified.
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.

Assigning Rules

Once you have created Rules, you can assign them to existing administrative objects that require them.

Each Attribute, Form, Program, and Relationship definition can refer to one Rule that encompasses owner, public and as many different users as required. The Rule becomes part of the definition of these kinds of objects.

Attributes, Forms, Programs, and Relationships created before version 6.0 of eMatrix have no Rules defined. By default, all existing definitions will still have no Rules, and will remain available to all users as before. However, use the procedures below if you want to add a Rule to an existing definition.

To add a Rule to an Attribute

Use the modify attribute command:

```
modify attribute ATTRIBUTENAME add rule RULE_NAME;
```

To add a Rule to a Form

Use the modify form command:

```
modify form FORMNAME add rule RULE_NAME;
```

To add a Rule to a Program

Use the modify Program command:

```
modify program PROGNAME add rule RULE_NAME;
```

To add a Rule to a Relationship

Use the modify Relationship command:

```
modify relationship RELNAME add rule RULE_NAME;
```

Deleting a Rule

If you decide that a rule is no longer required, you can delete it by using the Delete Rule statement:

```
delete rule RULE_NAME;
```

RULE_NAME is the name of the rule to be deleted. If the name is not found, an error message will result.

When this statement is processed, eMatrix searches the list of rules. If the name is found, that rule is deleted IF there are no objects that are governed by the rule. If there are objects that are governed by the rule, they must be reassigned to another rule or the object must be deleted before the rule can be removed from the rule list.

To remove a rule from a specific administrative object, use the Remove Rule statement on the object's modify command. For example, to remove a rule from an Attribute, use the `modify attribute ATTRIBUTENAME remove rule RULE_NAME;`

```
modify attribute ATTRIBUTENAME remove rule RULE_NAME;
```

Rules can be removed from attributes, forms, policies, relationships, and programs. Refer to [Assigning Rules](#) for additional information.

Working With Policies

Policy Defined	384
General Behavior.....	384
Lifecycle.....	384
Determining Policy States	385
How Many States are Required?	385
Who Will Have Object Access?	386
Is the Object Versionable or Revisionable?	386
How Do You Change From One State to the Next?	386
Defining an Object Policy	388
Description Clause.....	388
Icon Clause.....	389
Type Clause.....	389
Format Clause	390
Defaultformat Clause	391
Sequence Clause	391
Enforce Clause	393
State Clause	394
Store Clause	405
Hidden Clause	406
Property Clause	406
Copying and/or Modifying a Policy Definition	408
Copying (Cloning) a Policy Definition	408
Modifying a Policy Definition	408
Modifying Policy States.....	411
Modifying Signature Requirements.....	413
Deleting a Policy	415

Policy Defined

A *policy* controls a business object. It specifies the rules that govern access, approvals, lifecycle, revisioning, and more. If there is any question as to what you can do with a business object, it is most likely answered by looking at the object's policy.

You must be a Business Administrator to add or modify policies. (Refer also to your Business Modeler Guide.)

A policy is composed of two major sections: one describes the general behavior of the governed objects and the other describes the lifecycle of the objects.

General Behavior

The first section controls the creation of the object and provides general information about the policy. This information includes:

- The types of objects the policy will govern.
- The types of formats that are allowed.
- The default format automatically assigned.
- Where and how checked in files are managed.
- How revisions will be labeled.

Lifecycle

The second section provides information about the lifecycle of the objects governed by the policy. A lifecycle consists of a series of connected states, each of which represents a stage in the life of the governed objects. Depending on the type of object involved, the lifecycle might contain only one state or many states. The purpose of the lifecycle is to define:

- The current state of the object.
- Who will have access to the object.
- The type of access allowed.
- Whether or not the object can be revised.
- Whether or not files within the object can be revised.
- The conditions required for changing state.

Determining Policy States

When creating a policy, defining the policy states is most often the most difficult part. How many states does the policy need? Who should have access to the object at each state and what access should each person have at each state? Which access takes precedence over the other? Should you allow revisions at this state? Should you allow files to be edited? What signatures are required to move the object from one state to another? Can someone override another's signature? As described below, all of these questions should be answered in order to write the state definition section of a policy.

How Many States are Required?

A policy may have only one state or many. For example, you might have a policy that governs photographic images. These images may be of several types and formats, but they do not change their state. In general, they do not undergo dramatic changes or have stages where some people should access them and some should not. In this situation, you might have only one state where access is defined.

Let's examine a situation where you might have several states. Assume you have a policy to govern objects during construction of a house. These objects could have several states such as:

State	Description
Initial Preparation	The building site is evaluated and prepared by the site excavator and builder. After the site is reviewed and all preparations are completed, the excavator and builder sign off on it and the site enters the second state, Framing.
Framing	Carpenters complete the framing of the house and it is evaluated by the builder, architect, and customer. In this state, you may want to prohibit object editing so that only viewing is allowed. If the framing is complete to the satisfaction of the builder, architect, and customer, it is promoted to the third state, Wiring.
Wiring	The electrician wires the house. However, the electrician may sign off on the job as completed only to have the builder reject it. When approval is rejected, promotion to the next state is prevented from taking place.

As the house progresses through the building states, different persons would be involved in deciding whether or not the object is ready for the next state.

When determining how many states an object should have, you must know:

- What are the states in an object's life.
- Who requires access to the object.
- What type of access they need.

Once a policy is defined, you can alter it even after business object instances are created that are governed by it.

Who Will Have Object Access?

There are three general categories used to define who will have access to the object in each state:

- Public—refers to everyone in the eMatrix database. When the public has access in a state, any defined eMatrix user can work with the business object when it is in that state.
- Owner—refers to the specific person who is the current owner of the object instance. When an object is initially created in the database, the person who created it is identified by eMatrix as the *owner* of the object. This person remains the owner unless ownership is transferred to someone else.
- User—refers to a specific person, group, role, or association who will have access to the object. You can include or exclude selected groupings or individuals when defining who will have access.

For additional information on access privileges, including which access takes precedence over the other, see [User Access](#) in Chapter 10.

Is the Object Versionable or Revisionable?

In each state definition are the terms *Versionable* and *Revisionable*. The term *Versionable* in eMatrix indicates whether or not it is possible to check files into the object when it is in that state. The term *Revisionable* indicates whether a new Revision of the object can be made.

You can decide when in the object's lifecycle these operations are allowed by setting the switches ON or OFF in each state definition. These settings are independent of who (which person, role or group) has access to perform the operations.

How Do You Change From One State to the Next?

Most often a change in state is controlled by one or more persons, perhaps in a particular role or group. For example, during the construction of a house, the customer and the builder might control the change in state. If you break the building stage down into smaller states, you might have the object's transition controlled by the site excavator, foundation expert, electrician, or plumber. As the house progresses through the building states, different persons would be involved in deciding whether the object is ready for the next state. You certainly would not want the carpenters to begin working before the foundation is done.

In eMatrix, signatures are a way to control the change of an object's state. Signatures can be associated with a role, group, person, or association. Most often, they are role-related. When a signature is required, a person must approve the object in order for the object to move on to the next state. If that person does not approve it, the object remains in the current state until the person does approve or until someone with higher authority provides approval.

More than one signature can be associated with the transition of an object. Lifecycles can be set up such that the signature that is approved determines which state is the next in the object's life.

A signature can be approved or rejected. For example, an electrician could say a job is done only to have the builder reject it. When approval is rejected, promotion to the next state is prevented from taking place.

Filters can be defined on a signature requirement to determine if it is fulfilled. If the filter evaluates to true, then the signature requirement is fulfilled. This is useful for adding

required signatures that are conditional, dependent on some characteristic of a business object.

In the sections that follow, you will learn more about the actual procedures to define a policy and the object states as well as the procedures that manipulate and display policy definitions.

Defining an Object Policy

Policies are defined using the Add Policy statement:

```
add policy NAME [ ITEM {ITEM} ];
```

NAME is the name of the policy you are creating. All policies must have a name assigned. When you create a policy, you should assign a name that has meaning to both you and the user. The policy name is limited to 127 characters. For additional information, refer to *Business Object Name* in Chapter 34.

For example, you might have two policies that control the development of software programs for new and existing product lines. These policies might be named “Old Software Dev. Process” and “New Software Dev. Process.” While these are descriptive names, they are somewhat ambiguous. Does the “Old Software Dev. Process” identify a previously used process or a process for handling existing products? In this case, you could either include descriptions or assign more descriptive names such as “Existing Product Software Dev.” and “New Product Software Dev.”

ITEM is an Add Policy clause which defines information such as the types of objects governed by the policy, the types of formats permitted by the policy, the labeling sequence for revisions, the storage location for files governed by the policy, and the states and conditions that make up an object’s lifecycle. The Add Policy clauses are:

description VALUE
icon FILENAME
type TYPE_NAME { ,TYPE_NAME }
type all
format FORMAT_NAME { ,FORMAT_NAME }
format all
defaultformat FORMAT_NAME
sequence REVISION_SEQUENCE
[not]enforce
state STATE_NAME [STATE_ITEM { ,STATE_ITEM }]
store STORE_NAME
[! not]hidden
property NAME [to ADMINTYPE NAME] [value STRING]

Some of the clauses are required and some are optional, as described in the sections that follow.

Description Clause

The Description clause of the Add Policy statement provides general information about the types of rules associated with the policy. When a user creates a business object, this

description will guide him/her. Since there may be subtle differences between policies, the Description clause enables you to point out these differences.

There is no limit to the number of characters you can include in the description. However, keep in mind that the description is displayed when the mouse pointer stops over the policy in a chooser. Although you are not required to provide a description, this information is helpful when a choice is requested.

For example, assume you have two engineering development processes. One process is used by the Software Development group and the other is used by the Hardware Development group. These two processes might share some of the same states, types, formats and roles. However, the processes have different people involved as well as different requirements for development. You could document the differences in the Description clause:

```
add policy "Old Software Dev. Process"
    description "For developing enhancements for S/W";
add policy "New Software Dev. Process"
    description "For developing new standalone products";
```

Icon Clause

The Icon clause of the Add Policy statement associates an image with a policy. For example, you may have one policy for working with photographs and another for working with videos. You could assign an icon of a photograph to one and an icon of a video cassette to the other. The default icon is used when in view-by-icon mode. Any special icon you assign is used when in view-by-image mode. When assigning a unique icon, you must use a GIF image file. Refer to [Icon Clause](#) in Chapter 1 for a complete description of the Icon clause.

GIF filenames should not include the @ sign, as that is used internally by eMatrix.

Type Clause

The Type clause of the Add Policy statement defines all of the business object types governed by the policy. Just as a policy may govern many different object types, each object type may have many different policies that govern it. (However, an object instance can only have one policy associated at any time.)

For example, assume you have an object type named Drawing. This type may be governed by two policies named “Engineering Drawing Process” and “Documentation Drawing Process”. When an object of type Drawing is created, you must decide which policy will govern the object instance. A drawing meant for documentation will have a different review and lifecycle than a drawing of a component to an engineering assembly. By associating one policy with the created object, you control the types of files that can be checked in, who will use the object, and when it is used.

The Type clause is required for a policy to be usable:

```
type TYPE_NAME { , TYPE_NAME}
Or
type all
```

TYPE_NAME is a previously defined object type.

You can list one type or many types (separated by a comma or carriage return). When specifying the name of an object type, it must be of a type that already exists in the eMatrix database. If it is not, an error message will display.

For example, the following two statements are valid Add Policy statements that identify object types associated with the policy:

```
add policy "Engineering Revision Process"
    description "Quality Control Process for Engineering Revisions"
    type Drawing, "Multipage Drawings", Schematics, Manual;
add policy "Documentation Revision Process"
    description "Quality Control Process for Documentation Revisions"
    type Manual, "Release Notes";
```

In the first statement, the user can associate the “Engineering Revision Process” policy with object instances of four different object types: Drawing, Multi-page Drawings, Schematics, and Manual. In the second statement, the user can associate the “Documentation Revision Process” policy with only two object types: Manual and Release Notes. If the user created an object named “MQL User’s Guide” of type Manual, could he assign a policy of “Engineering Revision Process?” The answer is yes, although the “Documentation Revision Process” policy might be more appropriate.

A variation of the Type clause uses the keyword `all` in place of one or more type names (Type All). When this keyword is used, all of the business object types defined within eMatrix are allowed by the policy—the policy governs objects of all types. Use caution when including this clause. While all of the types currently defined may apply to the policy, what happens if a new one is created that should not apply?

If you have a policy that uses most of the defined objects, you may want to assign all of the types rather than list them. Then you can use the Modify Policy statement to remove the unwanted types.

Format Clause

The Format clause of the Add Policy statement defines the formats permitted under the policy for checked in files. Depending on the policy and the object created, certain files are appropriate or inappropriate. The Format clause restricts the types of files that can be associated with a business object.

The Format clause is required if you want to check in files under the policy:

```
format FORMAT_NAME { , FORMAT_NAME }
Or
format all
```

`FORMAT_NAME` is the name of a previously defined format. If the format was not previously defined, an error message results.

For example, you could have a policy that governs photos. This policy would need formats for processing files that contain photographic images. In this case, there may be two file formats allowed: GIF and JPG. These formats specify the software statements

required to view, edit, and print files of type Photo. You might write this policy definition as:

```
add policy Photo
  description "Photo Process"
  type Photo
  format GIF, JPG
  defaultformat GIF
  store Photo
  sequence "1,2,3,..."
  state base
  public all
  owner all;
```

Like the Type clause, the Format clause has a variation that uses the keyword `all`. When this clause is used, all of the formats defined within eMatrix are allowed under this policy—the policy governs objects of all formats. Use caution with the Format All clause. Are there any formats that you do not want to permit under this policy? If there are, you will need to either list all of the desired formats or edit the policy after assigning all formats.

Defaultformat Clause

The Defaultformat clause of the Add Policy statement is required in order to check any files in unless the Checkin clause specifies the format. If only one format is specified in the Format clause, it is automatically the default.

The Defaultformat clause has the syntax:

```
defaultformat FORMAT_NAME
```

`FORMAT_NAME` can be any previously defined format that is listed in the Format clause of the Add Policy statement. The Format clause identifies all formats permitted by the policy.

For example, the following Add Policy statement defines the default format as a text file that uses BestWord to process it:

```
add policy "Proposals"
  description "Process for generating, reviewing, and releasing proposals"
  type Proposal, Plan
  format ASCII, "BestWord", Drawing
  defaultformat "BestWord";
```

If an object does not have any files checked into its default format, execution of a View, Edit, or Print command will check its other formats and open files found there.

Sequence Clause

The Sequence clause of the Add Policy statement defines a scheme for labeling revisions. With this clause, you can specify the pattern to use when an existing object is revised. This pattern can include letters, numbers, or enumerated values. For example, you could have revisions labeled “1st Rev,” A, or 1.

To define a scheme for labeling revisions, you must build a revision sequence. This sequence specifies how objects should be labeled, the type of label to be used, and the

number of revisions allowed. When you create a revision sequence, use the following syntax rules:

Rule	Example
Hyphens denote range.	A-Z signifies that all letters from A through Z inclusive are to be used.
Commas separate enumerated types.	Rev1,Rev2,Rev3,Rev4 is a sequence with four revision labels. Rev1 will be assigned before Rev2, which will be assigned before Rev3, and so on.
Square brackets are used for repeating alphabetic sequences.	[A-Z] signifies that the sequence will repeat after Z is reached. When it repeats, it returns to the front of the label list and doubles the labels so that the next sequence is AA, AB, AC, and so on.
Rounded brackets are used for repeating numeric sequences.	(0-9) signifies a regular counting sequence. (When 9 is reached it will repeat and add a 1 before the symbol). If (0-5) was used, when 5 was reached it will repeat and add a 1: 0, 1, 2, 3, 4, 5, 10, 11, 12, etc.
A trailing ellipsis (...) means a continuing sequence.	A,B,C,... signifies the same thing as A-Z. 0,1,2,... signifies the same thing as (0-9)

These rules offer flexibility in defining the revision labeling sequence. Although you cannot have two repeating sequences in a single definition, you can include combinations of enumerated values and ranges within a repeating sequence. For example, the following revision sequence definition specifies that the first object should be labeled with a hyphen and the first revision should be labeled I, the second II, the third III, etc. After the fifth revision, all revisions will have numeric sequencing.

```
- ,I,II,III,IV,V,(0-9)
```

If your location requires a numeric value, for example, for pre-released revisions, and then an alphanumeric scheme after that, the approach should be to change the policy at the point when the revision scheme should change. A separate policy is created and applied to a new revision, providing different states, signatures, etc. as well as a different revision scheme. For example, if a revision sequence is defined as:

```
0,1,2,...,-,[A-H,J-N,P,R,T-W,Y]
```

the automatic sequencing will never get beyond the number counting, so the entries after that are ignored. Two policies should be established for the object type with revision sequences defined as follows:

```
0,1,2,...
```

and

```
- ,[A-H,J-N,P,R,T-W,Y]
```

If you enter blank spaces within the definition of a revision sequence, eMatrix uses the blank spaces literally. (In general, you should NOT use blank spaces within a revision sequence.) Consider the following examples:

Enter the revision sequence as:	eMatrix recognizes this as:
A, B, C	"A" " B" " C"
A,B,C,	"A" "B" "C"
1st Rev, 2nd Rev, 3rd Rev	"1st Rev" " 2nd Rev" " 3rd Rev"
1st Rev,2nd Rev,3rd Rev	"1st Rev" "2nd Rev" "3rd Rev"

After you define your revision sequence, simply insert it into the Sequence clause using the following syntax::

```
sequence REVISION_SEQUENCE
```

REVISION_SEQUENCE must follow the syntax rules given above.

For example, the following policy definition uses an enumerated revision sequence:

```
add policy "Engineering Proposal Process"
  type Proposal
  format Text
  sequence Unrevised,1st Rev,2nd Rev,3rd Rev,4th Rev;
```

In this statement, when a file named “Proposed Solar Vehicle” is checked into an object of type Proposal, it is named in the window as Proposal, “Proposed Solar Vehicle”, Unrevised. After the first revision, the object is given a revision label of “1st Rev” (in place of Unrevised). As it is further defined, eMatrix will progress through the enumerated sequence values until it reaches “4th Rev.” Since this sequence is non-repeating, no further revisions are allowed.

Enforce Clause

The enforce clause is optional and can be used to prevent one user from overwriting changes to a file made by another user. When an object is governed by a policy that has enforce locking turned on, the only time a user can check in files *that replace existing files* is when:

- the object is locked
- and
- the user performing the checkin is the locker

To ensure that the person who locked the object is the person who checked out the file, enforce locking disables the manual lock function (`lock businessobject OBJECTID;`). The only way to lock an object that is governed by a policy that enforces locking is by locking it when checking out the file (for example: `checkout bus OBJECTID lock;`).

When checking in files *that do not replace existing files* (for example, if you check files into a format that contains no files or you append files), as long as the object is unlocked, you can check in new files. When an object is locked, no files can be checked into the object until the lock is released, even if the file does not replace the checked-out file that initiated the lock. This means that attempts to open for editing, as well as checkin, will fail. Files can be checked out of a locked object and also opened for viewing.

Enforce locking ensures that when a user checks out a file and locks the object, signifying that the user intends to edit the file, no other user can check in a file and overwrite the files the original user is working on. When the original user checks the file back in, the user should unlock the object.

Be aware that the manual unlock command (`unlock businessobject OBJECTID;`) is available for users who have unlock access, but users should avoid using the command for objects that have enforce locking. For example, suppose Janet checks out a file and locks the object with the intention of editing the file and checking it back in. Steve, who has unlock access, decides he needs to check in an additional file for the object so he unlocks the object manually. When Janet attempts to check in her edited file, replacing the original with her updated file, the system won't allow her to because the object isn't locked. In order to check in the file, Janet has several options:

- she can check in the edited file in such a way that it won't replace existing files; for example, change the name of the edited file and append it or delete the original file and check in the edited file
- she can check out the original file again and lock the object, taking care not to replace the edited file on her hard drive with the older file she is checking out, and then check in the edited file

For more information on unlock access, see [Accesses](#) in Chapter 10.

A user would end up in a situation similar to the one described above if the user forgets to lock the object when checking out a file for editing. When the user attempts to check in the edited file, the system won't allow the checkin because the object is unlocked (or possibly locked by another user who checked out the file after the first user).

For example, to enforce locking on the Proposals policy:

```
add policy "Proposals"
description "Process for generating, reviewing, and releasing proposals"
type Proposal, Plan
format ASCII, "BestWord", Drawing
defaultformat "BestWord"
enforce;
```

The `not enforce` clause is available when modifying policies to turn the feature off.

State Clause

The State clause of the Add Policy statement defines all information related to a policy state including: who can access a business object, what type of access a user can have, whether new revisions are allowed, and the conditions for changing from one state to another. The State clause uses the following syntax:

```
state STATE_NAME [STATE_ITEM {,STATE_ITEM}]
```

`STATE_NAME` is the name of the state you are defining. All states must have a name assigned. This name must be unique within the policy and should have meaning for both you and the user. The state name is limited to 127 characters. For additional information, refer to [Business Object Name](#) in Chapter 34.

For example, assume you have a process for performing and evaluating lab tests. The first state might involve receiving the initial test request, gathering information on the item or person to be tested, and getting approval for the test. This state could be called “Initial Test Processing” or “Test Request.” Once the testing is approved, the test object might enter a second state where the test is actually performed. This state could be called the “Testing,” “Actual Test Processing,” or “Lab Work” state. After the test is completed, the object might then be available for evaluation and review. This final state could be called the “Test Results,” “Test Evaluation,” or “Test Review” state. In each example, the names provide some indication of what is happening to the test object in each state.

STATE_ITEM is a State subclause which provides additional information about the state. The State definition subclauses are:

action COMMAND
check COMMAND
notify USER_NAME { ,USER_NAME } message VALUE
notify signer message VALUE
route USER_NAME message VALUE
owner ACCESS_ITEM { ,ACCESS_ITEM}
public ACCESS_ITEM { ,ACCESS_ITEM}
user USER_NAME ACCESS_ITEM { ,ACCESS_ITEM} [filter EXPRESSION]
signature SIGN_NAME [SIGNATURE_ITEM { ,SIGNATURE_ITEM}]
trigger EVENT_TYPE TRIGGER_TYPE PROG_NAME [input ARG_STRING]
revision [true false]
version [true false]
promote [true false]
icon FILENAME
checkouthistory [true false]

When defining a policy, you must have at least one state defined. Within that state, you must define some type of object access. All other information is optional. The sections that follow describe these clauses and the arguments they use.

Action Subclause

The Action subclause of the State clause associates a program with the promotion of this state. Once an object is promoted to this state, eMatrix executes the program specified by the clause. (Refer to the example on the next page.)

Although the Action subclause is optional, it is useful for executing procedures that might notify non-eMatrix users, generate reports, or place orders for equipment or services:

action PROGRAM

PROGRAM is the name of a program object, or method, that has been or will be defined by the Business Administrator.

Check Subclause

The Check subclause of the State clause associates a verification procedure with the promotion of the object out of the state. The procedure is specified as a program which is executed when a person tries to promote the object. When executed, the procedure returns a true or false value. If the value is true, the object is promoted. If the value is false, the promotion is denied. Refer also to [Overview of Programs](#) in Chapter 19.

Use the following syntax to write a Check subclause:

```
check PROGRAM
```

PROGRAM is the name of a program object, or method, that has been or will be defined by the Business Administrator.

Notify Subclause

The Notify Subclause sends a message to selected users once a business object has entered the state. The message might provide special instructions or notify users that an object is ready for a particular action. The notification message is limited to 255 characters.

Use the following syntax to write an Notify subclause:

```
notify USER_NAME { ,USER_NAME } message VALUE
```

Or:

```
notify signer message VALUE
```

USER_NAME is the name of a person, group, role, or association to notify.

Signer refers to the users who are included in the signature requirements for the next state. This allows one notify message to notify all signers automatically.

VALUE is the text of the message to be sent to the user(s).

For example, assume you have a user manual that is being written. In its beginning state, only the author and the author's manager might access the document. However, once the manual is ready for review, it would most likely be promoted to a state where it is available to other users for comments. When this occurs, users must be notified that the manual is available and that review comments are required. The following subclause notifies two groups that a manual is ready for review:

```
notify Engineering, Training  
message "The User's Guide is now ready for review. Please have your review comments  
completed in two weeks."
```

Example of Action, Check, and Notify

Assume that the states of the policy governing a type "Solid Model" are:

Planned —> Started —> Ready for Detail —> Released

Also assume that the states of the policy for Drawing objects are:

Planned —> Submit for Check —> Checked —> Released

When a solid model reaches the "Ready for Detail" state, the drafter begins work on the drawing. An action could be declared at this state to execute a program to create a

Drawing business object of the same name as the solid model object. The program also could connect the two objects with the “Solid-Drawing” relationship. When arriving in this state, eMatrix could send IconMail (using “notify” in the state definition) to the drafting manager explaining that the solid model is ready for detail and that drawing files should be checked into the attached Drawing object.

Before the drawing is released, a check should be performed to be sure the solid model was released. The check would be declared in the transition arrow before the “Released” state of the Drawing policy. The check would execute a program that would expand the object connected by the “Solid-Drawing” relationship and check their current state. If the current state is “Released,” promotion of the Drawing to the “Released” state would be allowed; otherwise, promotion would fail.

Route Subclause

The Route subclause automatically reassigns ownership when an object enters a business state. Since ownership implies greater access privileges and responsibility, changing ownership can be an effective means of controlling an object. The Route subclause is also used to notify the new owner of the reassessments. The route message is limited to 255 characters.

For example, assume you have a manual that was just completed by a writer. The manual is promoted into a state called “Formatting and Editing” in which an editor takes charge of the manual. The editor prepares the document for review and oversees any changes required to prepare it for publication. Since the writer is no longer involved, you may want to assign the manual’s ownership to the editor. While the editor might be among the users notified that the manual is finished, the editor should receive special notification that the manual now “belongs” to him/her. After the manual is published, you might again change the ownership to that of the company librarian. When changes in ownership occur, the new owner should be notified.

The Route subclause is an optional subclause that is very similar to the Notify subclause (described in [Notify Subclause](#)).

```
route USER_NAME { ,USER_NAME } message VALUE
```

USER_NAME is the name of a person, group, or role who will receive ownership of the business object.

VALUE is the text of the message to be sent to the new owner(s)

For example, the following subclause notifies the editor that ownership of a manual was transferred to him/her:

```
route Editor  
message "Ownership of this manual has been transferred to you."
```

Owner, Public, and User Subclauses

The Public, Owner, and User subclauses of the State clause define who will have access to the business objects and what type of access they will have. When you are defining a policy, you must have at least one state defined. Within that state you should include at least one of these subclauses if the object is to be accessible.

As described in [Who Will Have Object Access?](#) at the beginning of this chapter, Public refers to everyone in the eMatrix database, Owner refers to the person who currently owns the business object or to whom the object was routed or reassigned, and User can be

defined as any eMatrix group, role, or person who is using the business object. In the Public, Owner, and User subclauses, you are essentially defining the object access:

```
owner ACCESS_ITEM { ,ACCESS_ITEM}  
public ACCESS_ITEM { ,ACCESS_ITEM}  
user USER_NAME ACCESS_ITEM { ,ACCESS_ITEM} [filter EXPRESSION]
```

ACCESS_ITEM specifies an access privilege to be associated with the access category. Each access form (or access *mask*) is used to control some aspect of a business object's use.

USER_NAME is the name of a person, group, role, or association to have access.

EXPRESSION is a filter expression defined for the user access. Expression access filters can be any expression that is valid in the eMatrix system. Expressions are supported in various modules of the eMatrix system, for example, query where clauses, expand, filters defined on workspace objects such as cues, tips and filters, etc. See [Working With Expression Access Filters](#) in Chapter 10 for details on how access filters can be used.

```
add policy Production1  
    state Started  
    owner read  
    public none  
    user Training read modify checkin  
        filter attribute[Target Weight] == 35.2;
```

Here a filter Expression is being applied to the User Access called "Training" in the state "Started" of the policy named "Production1".

Without the expression access filter, the selected access (read, modify and checkin) would have been allowed:

- if context user belongs to the group "Training" and
- business object being accessed is in the state "Started"

With this feature, the selected access (read, modify, checkin) would only be allowed:

- if context user belongs to the group "Training" and
- the business object being accessed is in the state "Started" and
- if filter expression (attribute[Target Weight] == 35.2) is evaluated as true for the business object being accessed.

Note that in this situation "Target Weight" is an attribute of the business object being accessed. If the "Target Weight" attribute doesn't exist for the business object being accessed, the read, modify and checkin access would not be allowed for the context user via this user access.

Since you must have at least read access to the business object in order to evaluate the filter, normal access checks must be disabled while an access filter is being evaluated.

The ACCESS_ITEMS are described in the table below. With each ACCESS_ITEM, other than all and none, you can either grant the access or explicitly deny the access. You deny an access by entering not (or !) in front of the ACCESS_ITEM in the statement.

Summary of Access Items	
Access Item	Allows a user to:
READ	View the properties of an object. Refer to <i>A note about Read Access</i> , below.
MODIFY	Edit the object.
DELETE	Delete the object from the database.
CHECKOUT	Copy files contained within a business object to the local workstation.
CHECKIN	Copy files from the local workstation to a business object.
SCHEDULE	Set and modify schedule dates for a business object.
LOCK	Restrict other users from checking files into a business object.
UNLOCK	Release a lock placed upon a business object <i>by another user</i> . Users may release locks which they themselves have placed on objects without this access.
FREEZE	Freeze, or lock, a relationship so that business objects may not be disconnected until the relationship is thawed. Also, the type or attributes of a frozen relationship may not be modified.
THAW	Thaw, or unlock, a relationship so that it may be modified or deleted.
CREATE	Create business objects or relationships.
REVISE	Create a revision of a selected business object.
GRANT	Grant privileges to another user.
REVOKE	Revoke privileges granted to another user.
PROMOTE	Change the state of an object to be that of the next state.
DEMOTE	Change the state of an object to that of a prior state.
ENABLE	Unlock the state so that a business object can be promoted or demoted.
DISABLE	Lock a state so that a business object cannot be promoted or demoted.
OVERRIDE	Promote an object even when the conditions for changing the state have not been met.
CHANGENAME	Change the name of a business object.
CHANGETYPE	Change the type of a business object.
CHANGEOWNER	Change the owner of a business object.
CHANGEPOLICY	Change the policy of a business object.
CHANGEVAULT	Change the vault of a business object.

Summary of Access Items	
Access Item	Allows a user to:
FROMCONNECT	Link business objects together on the "from" side of a relationship.
TOCONNECT	Link business objects together on the "to" side of a relationship.
FROMDISCONNECT	Dissolve a relationship on "from" business objects.
TODISCONNECT	Dissolve the "to" side relationship between business objects.
<i>Note: Object connect and disconnect access privileges are checked on both the from and to objects for the appropriate to/from privilege. Create or delete access must also be granted on the relationship type.</i>	
MODIFYFORM	Edit attribute and other field values in a form. Must also have modify access on the Attribute being modified.
VIEWFORM	Read a form.
EXECUTE	Execute a program. Includes programs specified as View, Edit, and Print commands in Formats; check, override and action programs in Triggers; and Business Wizards.
SHOW	Control whether a user knows that a business object exists.
You can type not or ! at the beginning of an access mask to explicitly deny the access; for example, !todisconnect or notchangeowner.	

Access privileges can be assigned using one of two methods:

- *Inclusion*, which assumes that you are starting with no privileges. You then list all of the privileges that will be included in this access list.
- *Exclusion*, which assumes that you are starting with all privileges. From this list, you then want to exclude specific access privileges from that user.

The method you use will depend on the amount of access privileges you want to assign. If only a few privileges are to be assigned, use the inclusion method. If most of the privileges are to be assigned, use the exclusion method.

For example, assume you had the following state definition:

```
state Reviewed
    revision false
    version true
    public checkout, read
    owner all, notpromote
    user "Product Group Supervisor" changeowner, promote
```

In this definition, each user has a different level of access. As defined in the Public subclause, the public has checkout access only when an object is in this state. This is an example of the inclusion method of defining access. While the keyword "none" is not specified, it is implied and is used as the default value. Another example of the inclusion method is seen in the User subclause. The "Product Group Supervisor" is assigned the ability to reassign ownership and to promote the object to the next state. No other privileges have been assigned. Now examine the Owner subclause. It shows an example of the exclusion method for assigning access. The Owner is assigned all privileges and

then excluded from using the promote privilege. This means the Owner will not be able to promote the object to the next state. That privilege is reserved for the supervisor.

When you define a state, you should define some access for one of the three access categories. If you are unsure of what access to assign, use the Public subclause. This will allow any user in the system to create, modify, and manipulate an object under the policy.

If you do not want the public to have complete access over an object you create, you should include an Owner subclause in the state definition. Then you can restrict public access while allowing the owner complete control. For example, the following allows the public to create and read objects only, while the owner has complete access:

```
state Proposed
    revision false
    version true
    public create, checkout
    owner all
    user "Product Group Supervisor" changeowner, promote
```

In this example, a User clause is included to allow a change of ownership. This allows the supervisor to reassign the object to another person if the original owner leaves. Also, the supervisor can change the state of the object to the next state. While create and checkout are not listed under the User subclause, the supervisor is a member of the eMatrix database public and, therefore, has those privileges.

Signature Subclause

The Signature Subclause of the State clause specifies who can control the promotion or rejection of a business object. When an object is promoted, it moves to the next defined state and is subject to the access rules associated with that state. When an object is rejected, it remains in the current state until it meets the criteria for promotion.

The Signature subclause has the syntax:

```
signature SIGN_NAME [SIGNATURE_ITEM {,SIGNATURE_ITEM}]
```

SIGN_NAME identifies the type of signature. Try to use a name that identifies what the signature represents. For example, the signature might represent initial acceptance, completion, or final sign-off. Each of these terms could be used for a SIGN_NAME. The signature name is limited to 127 characters. For additional information, refer to *Business Object Name* in Chapter 34.

SIGNATURE_ITEM identifies the type of state change that will occur when a group, role, or person signs off on an object. Use any of the following:

approve USER_NAME {,USER_NAME}	Specifies that the object may be promoted to the next state.
reject USER_NAME {,USER_NAME}	Specifies that the object must remain in the current state until it meets with the approval of the user.
ignore USER_NAME {,USER_NAME}	Enables you to override the approval or rejection of the object. When ignore is specified, the named user can sign in the place of others. This might be useful to allow a senior manager the ability to sign off for a lower manager.

branch STATE_NAME	Specifies what the next state will be after a signature is applied.
filter EXPRESSION	Enables filtering to ensure that the promotion of an object meets certain criteria.

STATE_NAME is any previously-defined name of a state included in the current policy. By specifying a branch, you can decide which signatures are required to transition to a given state during a promote operation. When promotion is initiated, the system will choose the state for which all signatures are satisfied. If more than one branch is enabled, an error is generated.

EXPRESSION is a statement that evaluates to either true or false. If a signature requirement *filter* evaluates to true, then the signature requirement is fulfilled. This is useful for adding required signatures that are conditional, dependent on some characteristic of a business object. The default rule is:

```
current.signature[NAME].satisfied
```

which is a select field that means the signature has been approved, ignored, or overridden. When you specify a filter, this default rule is replaced.

Approval can become dependent on any selectable field of the business object. This includes attributes as well as states and other signatures. The real power of filters comes from the use of combinatorial logic using and's and or's between state information and business object information.

For help formatting the expressions that can be entered into the Filter area, see *Appendix A, Select Expressions* in the *eMatrix Navigator Guide*.

USER_NAME is any previously defined name of an eMatrix association, group, role, or person. If the name you give is not defined, an error message will result. If you are unsure of a user name, remember that you can obtain a complete listing of all of the user names by entering the MQL List User statement.

For example, assume you have a state where objects are started and worked on prior to general review. While the object is in this state, you may want two types of signatures: one to indicate that the project is complete and another to indicate acceptance. This state definition might appear as:

```
state started
    revision false
    public all, notenable, notdisable, notoverride
    owner all, notenable, notdisable, notoverride
    user Manager override
    signature Complete
        approve Writer
        reject Writer
        ignore Manager
    signature Accepted
        approve Manager
        reject Manager
        ignore "Senior Manager"
```

When including an Ignore Signature item in a state definition, you should not confuse this with the override privilege. The Override privilege allows you to promote an object without any signatures at all. The Ignore Signature item assumes that a signature is required for object promotion. It simply allows the specified person to provide that required signature.

When a signature has been approved, it can subsequently be rejected. But when a signature has been rejected, ignore cannot subsequently be used to satisfy the signature.

Trigger Subclause

Event Triggers allow the execution of a Program object to be associated with the occurrence of an event. The following lifecycle events support Triggers:

approve	demote	disable
enable	ignore	override
promote	reject	schedule
unsign		

For example, when a state was scheduled, each successive state could be scheduled automatically by a specified offset value. These transactions are written into program objects which are then called by the trigger.

State Triggers use the following syntax:

```
trigger EVENT_TYPE TRIGGER_TYPE PROG_NAME [input ARG_STRING];
```

EVENT_TYPE is any of the valid events for Policies: approve, demote, disable, enable, ignore, override, promote, reject, schedule, or unsign.

TRIGGER_TYPE is Check, Override, or Action. Refer to [Types of Triggers](#) in Chapter 20.

PROG_NAME is the name of the Program object that will execute when the event occurs.

ARG_STRING is a string of arguments to be passed into the program. When you pass arguments into the program they are referenced by variables within the program. Variables 0, 1, 2... etc. are reserved by the system for passing in arguments.

Environment variable “0” always holds the program name and is set automatically by the system.

Arguments following the program name are set in environment variables “1”, “2”,... etc.

See [Using the Runtime Program Environment](#) in Chapter 19 for additional information on program environment variables.

For example:

```
state started
    revision false
    public all, notenable, notdisable, notoverride
    owner all, notenable, notdisable, notoverride
    user Manager override
    signature Complete
        approve Writer
        reject Writer
        ignore Manager
    signature Accepted
        approve Manager
        reject Manager
        ignore "Senior Manager"
trigger schedule action "Schedule Offsets";
```

Refer to [Designing Triggers](#) in Chapter 20, for more information on designing Triggers.

Revision Subclause

The Revision subclause of the State clause specifies whether or not revisions of the object are allowed while the object is in this state.

This subclause uses two arguments: true and false.

```
revision [true|false]
```

When the revision argument is set to `true`, revisions of the object are allowed. If the revision argument is set to `false`, no revisions are allowed.

For example, the following Revision subclause prohibits the creation of a new revision while the object is in this state::

```
state "Tax Return Completed"
    revision false
    public none, read
    owner none, read, demote
    user Manager none, read, promote
```

In this example, you have a state for completed tax forms. In this state, you do not want anyone to revise the objects containing the finished tax returns. If further modification is required, a change in state must occur. The owner can demote the object to its former state to make changes and the manager can promote the object into the audit state.

The Revision subclause is optional. eMatrix will assume that revisions are allowed if no subclause or argument is used. Use the `false` keyword to turn off the ability to create revisions.

Version Subclause

The Version subclause of the State clause indicates whether or not any file can be checked in while the object is in the state.

Like the Revision subclause described above, this subclause uses two arguments: true and false.

```
version [true|false]
```

When the version argument is set to `true`, files can be checked in. If the argument is set to `false`, no new files are allowed.

In the following state definition, the Customer or the builder/owner can check in files (that might contain room layouts, exterior views, or electrical plans, for example).

```
state "House Design Phase"
    revision false
    version true
    public none, read
    owner all
    user Customer none, read, checkin, modify
```

The Version subclause is optional. eMatrix assumes files can be checked into the object while the object is in the state if no subclause or argument is used. Use the `false` keyword to turn off the ability to check in files.

Promote Subclause

The Promote subclause of the State clause specifies whether or not eMatrix will automatically test the business object for promotion.

This subclause uses two arguments: true and false.

```
promote [true|false]
```

If the keyword `true` is used and all signature and check requirements are satisfied, eMatrix will promote the business object automatically. The Promote subclause is optional. eMatrix will assume that promotions are not automatic if no subclause or argument is used. Use the `false` keyword to turn auto-promotion off.

Icon Subclause

The Icon Subclause of the State clause associates a special icon with a state. While this subclause is optional, it can assist a user in identifying a policy state. The user could tell at a glance which state the object is in when viewing the object's lifecycle in a window.

Checkouthistory Subclause

The generation of history information on the checkout event is optional. The need to disable checkout history stems from the implementation of distributed databases. Creating history records requires that a distributed transaction be run across multiple servers. If any server is unavailable, the transaction will fail. This means that all servers must be available in order to checkout/view files. If checkout history is disabled, only the local server needs to be accessible in order for the transaction to run to completion. This subclause uses two arguments: true and false.

```
checkouthistory [true|false]
```

Store Clause

The Store Clause of the Add Policy statement identifies where files checked in under the policy are stored. All files must be stored as ingested, captured, or tracked files. (For more information on file stores, see [Store Defined](#) in Chapter 5.) If you intend to associate files with business objects that are governed by the policy, you must include a Store clause in the policy definition:

```
store STORE_NAME
```

`STORE_NAME` is the name of a previously defined file store. If the name you provide is not defined, an error message will result.

For example, assume you have a policy for proposing and presenting drawings for review. These drawings may be of various types and formats. However, all information about the drawings can be contained in one file store. This file store identifies how the drawing files

are managed and where they are stored. If you were to examine this policy, it might appear similar to the following:

```
add policy "Proposed Drawings"
description "Policy for Drawing Proposal and Presentation"
type Drawing, Layout, Schematic, Sheet
format Cadra-III, Rosetta-preView, CCITT-IV
defaultformat Cadra-III
sequence "A,B,C,..."
store Drawings
state planned
public all
owner all
state started
public all, notenable, notdisable, notoverride
owner all, notenable, notdisable, notoverride
user Employee enable, disable
user Manager override
signature Complete
approve Manager
reject Employee
ignore Designer
signature Accepted
approve Manager
reject Manager
ignore Manager
state presented
public all, notdelete
owner all;
```

In this example, the policy governs four types of business objects and uses three different file formats. However, when a file is checked into a business object governed by this policy, that file is managed and stored according to the definition of the Drawings file store.

MQL users and programmers may override the store specified in the policy, by including the store in the checkin command.

For information on changing the store for a policy, see [Modifying a Policy Definition](#).

Hidden Clause

You can specify that the new policy is “hidden” so that it does not appear in the Policy chooser in eMatrix. You may want to use the hidden option if, for example, an object is under development or if it is intended only for your personal use. Users who are aware of the hidden policy’s existence can enter its name manually where appropriate. Hidden objects are also accessible through MQL.

Property Clause

Integrators can assign ad hoc attributes, called Properties, to the policy. Properties allow associations to exist between administrative definitions that aren’t already associated. The property information can include a name, an arbitrary string value, and a reference to another administration object. The property name is always required. The value string and object reference are both optional. The property name can be reused for different object

references, that is, the name joined with the object reference must be unique for any object that has properties.

```
add policy NAME  
    property NAME [to ADMINTYPE NAME] [value STRING];
```

For additional information on properties, see *Overview of Administration Properties* in Chapter 24.

Copying and/or Modifying a Policy Definition

Copying (Cloning) a Policy Definition

After a policy is defined, you can clone the definition with the Copy Policy statement. This statement lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy policy SRC_NAME DST_NAME [MOD_ITEM] { ,MOD_ITEM} ;
```

SRC_NAME is the name of the policy definition (source) to copied.

DST_NAME is the name of the new definition (destination).

MOD_ITEMS are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

Modifying a Policy Definition

After a policy is defined, you can change the definition with the Modify Policy statement. This statement lets you add or remove defining clauses and change the value of clause arguments:

```
modify policy NAME [MOD_ITEM] {MOD_ITEM} ;
```

NAME is the name of the policy you want to modify.

MOD_ITEM is the type of modification you want to make. Each is specified in a Modify Policy clause, as listed in the following table. Note that you only need to specify fields to be modified.

Modify Policy Clause	Specifies that...
defaultformat FORMAT_NAME	The default is set to the format named.
description VALUE	The current description, if any, is changed to the values entered.
icon FILENAME	The image is changed to the new image in the file specified.
add type TYPE_NAME { ,TYPE_NAME }	The named types are added to the list of object types that can have this policy.
add type all	All object types are permitted under this policy.
remove type TYPE_NAME { ,TYPE_NAME }	The named object types are removed from the list of objects that can have this policy.
remove type all	All object types are removed from the list of object types permitted by this policy. If no types are allowed, no objects may be created under this policy.
add format FORMAT_NAME { ,FORMAT_NAME }	The named formats are added to the list of formats permitted by this policy.
add format all	All format types are permitted with this policy.
remove format FORMAT_NAME { ,FORMAT_NAME }	The named formats are removed from the list of formats permitted by this policy.

Modify Policy Clause	Specifies that...
remove format all	All formats are removed from the list of formats permitted by this policy.
add state STATE_NAME [before STATE_NAME] [STATE_ITEM { ,STATE_ITEM}]	<p>The named state is added to the policy with the state definitions listed. If you do not want the new state added after the existing states, you must specify which existing state the new state should precede.</p> <p>If a state is added to an existing policy which already governs objects, all object instances will be affected.</p> <p>If an object is in a state that precedes the new state, a state is added, as desired, in the object's lifecycle. However, if the object's current state is beyond where the new state is added, the object will never reach that state except through demotion. In some cases, this is not a concern; but, states should be added to existing policies with care.</p>
remove state STATE_NAME	<p>The named state is removed from the policy if there is at least one state remaining after the removal. Removing a state from a policy that is governing objects is not recommended.</p> <p>An alternate approach is to clone the policy and then remove the state from the clone. There is the notion that "from this point on" the policy will control these types of objects. New objects should use the new policy and older objects can change to the new policy, if desired.</p>
name VALUE	The current state name is changed to that of the new name.
sequence REVISION_SEQUENCE	The revision sequence is changed to the sequence entered.
state STATE_NAME [STATE_MOD_ITEM {STATE_MOD_ITEM}]	The named state is changed according to the state modification clauses entered.

Modify Policy Clause	Specifies that...
store STORE_NAME	<p>The file store is changed to use the file store named.</p> <p>Keep in mind that if you change the store for a policy, files that are already checked into objects governed by the policy will still reside in the old store. If these objects are revised or cloned, the new revision/clone references the original file in its storage location and thus the clone or revision will be placed in the old storage location. When the time comes for the file reference to become an actual file (as when the file list changes between the 2 objects) the file copy is made in the same store the original file is located in.</p> <p>However any new files that are checked in will be placed in the new store. For more details, see the <i>Implications of Changing Stores</i> in Chapter 6.</p>
checkouthistory true	The generation of history information on the checkout event is enabled.
checkouthistory false	The generation of history information on the checkout event is disabled.
hidden	The hidden option is changed to specify that the object is hidden.
nothidden	The hidden option is changed to specify that the object is not hidden.
property NAME [to ADMINTYPE NAME] [value STRING]	The named property is modified.
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.

Each modification clause is related to the clauses and arguments that define the policy. For example, assume you want to modify the policy for proposing, presenting, and releasing drawings. It has been decided to use the policy for only CAD drawings while a new policy is defined for handling non-CAD drawings. To customize the existing policy for CAD use, you might begin modifying the name and the policy's description clause with the following Modify Policy statement:

```
modify policy Drawings
  name "CAD Drawings"
  description "Policy for CAD Drawing Proposal, Review and Release";
```

These changes leave only the states to be modified. But how is that done? Just as the Modify Policy clauses resemble the Add Policy clauses, the subclauses that modify states resemble those that define them. These are described in the following sections.

Modifying Policy States

The following subclauses are available to modify the existing states in a policy:

Modify Policy State Subclause	Specifies that...
action COMMAND	The action defined by the command is taken when the object is promoted to this state.
check COMMAND	The verification test to be executed when the object is promoted to this state changes as specified by the command. This test must return a true or false value.
icon FILENAME	The icon file name associated with this state, if any, is set to the value entered.
name VALUE	The name of the current state is changed to the new name entered.
add notify USER_NAME { ,USER_NAME } [message VALUE]	The user(s) listed is/are added to those notified when the object is promoted to this state. If a message value is entered, the message changes to the new value.
remove notify USER_NAME { ,USER_NAME }	The user(s) listed is/are removed from the list of users who are notified when the object is promoted to this state.
add notify signer [message VALUE]	The user(s) included in the signature requirements for the next state is/are notified when the object is promoted to the state. This allows one notify message to notify all signers automatically.
remove notify signer	The user(s) included in the signature requirements for the next state are removed from the list of users who are notified when the object is promoted to the state.
promote false	Promote to the state will not occur automatically even if all conditions are met.
add owner ACCESS_ITEM { ,ACCESS_ITEM}	One or more access items are added to the owner access list. This list specifies the access privileges the owner has when the governed object is in this state.
remove owner ACCESS_ITEM { ,ACCESS_ITEM}	The listed access item is removed from the owner access list.
add public ACCESS_ITEM { ,ACCESS_ITEM}	One or more access items are added to the public access list. This list specifies the access privileges the public has when the governed object is in this state.
remove public ACCESS_ITEM { ,ACCESS_ITEM}	The listed access item is removed from the public access list.
revision true	Revisions are allowed in this state.
revision false	Revisions are not allowed in this state.

Modify Policy State Subclause	Specifies that...
add route USER_NAME { ,USER_NAME } [message VALUE]	The user(s) listed is/are added to those who will receive ownership of the business object when the object is promoted to this state. If a message value is entered, the message changes to the new value.
remove route USER_NAME { ,USER_NAME }	The user(s) listed is/are removed from the list of users who receive ownership of the object when it is promoted to this state.
route message VALUE	The message sent to users who receive ownership of the business object when the object is promoted to this state changes to the value entered.
add signature SIGN_NAME [SIGNATURE_ITEM { ,SIGNATURE_ITEM}]	The name(s) listed can promote, reject, or ignore the business object.
remove signature SIGN_NAME	The entered signature is removed from the list of signatures required to alter the object's state.
signature SIGN_NAME [SIGNATURE_MOD_ITEM {SIGNATURE_MOD_ITEM}]	The named signature is modified according to the modification clause(s) listed.
add trigger EVENT_TYPE TRIGGER_TYPE PROG_NAME	The specified trigger is added or modified for the listed event.
remove trigger EVENT_TYPE TRIGGER_TYPE PROG_NAME	The specified trigger type is removed from the listed event.
add user USER_NAME ACCESS_ITEM { ,ACCESS_ITEM}	One or more access items are added to the user access list. This list specifies the access privileges the named user has when the governed object is in this state.
remove user USER_NAME ACCESS_ITEM { ,ACCESS_ITEM}	The listed access item(s) is/are removed from the named user's access list.
version true	New versions are allowed in this state.
version false	New versions are not allowed in this state.

When modifying the states of a policy, the state modification clauses are similar to those used to define states. For example, assume you had the following state definition:

```
state "In Progress"
revision false
version true
public all
owner all
```

Now you want to have a Manager oversee the work in progress. Rather than let the owner or public promote the object into the next state, you want to give that privilege to the Manager only. To make these changes, you might write a statement such as:

```
modify policy "Manual Release"
  state "In Progress"
    add public notenable, notdisable, notoverride, notpromote
    add owner notenable, notdisable, notoverride, notpromote
    add user Manager promote, enable, disable, override;
```

When this statement is processed, the state definition is modified to appear as:

```
state In Progress
  revision false
  version true
  public notenable, notdisable, notoverride, notpromote
  owner notenable, notdisable, notoverride, notpromote
  user Manager promote, enable, disable, override;
```

Modifying Signature Requirements

If you want to alter the signature requirements within a state, you must use a Signature subclause within the State subclause:

```
signature SIGN_NAME [SIGNATURE_MOD_ITEM {SIGNATURE_MOD_ITEM}]
```

SIGN_NAME is the name of the signature item to modify.

SIGNATURE_MOD_ITEM identifies the type of modification.

You can make the following types of modifications to a signature clause. These types of modifications are related to the subclauses you saw in defining the signature requirements:

Modify Signature Subclause	Specifies that...
add approve USER_NAME {,USER_NAME}	The users are added to the list of people who can approve of the object.
remove approve USER_NAME {,USER_NAME}	The users are removed from the list of people who can approve of the object.
add ignore USER_NAME {,USER_NAME}	The users are added to the list of people who can sign in place of an approver or rejecter.
remove ignore USER_NAME {,USER_NAME}	The users are removed from the list of people who can sign in place of an approver rejecter.
add reject USER_NAME {,USER_NAME}	The users are added to the list of people who can reject the object.
remove reject USER_NAME {,USER_NAME}	The users are removed from the list of people who can reject the object.
add branch STATE_NAME	The branch is added to the signature.
remove branch STATE_NAME	The branch is removed from the signature.
add filter EXPRESSION	The filter is added for the signature.
remove filter EXPRESSION	The filter is removed from the signature.

The following state definition creates user documentation:

```
state "In Progress"
    revision false
    version true
    public all, notenable, notdisable, notoverride
    owner all, notenable, notdisable, notoverride
    user Manager override
    signature Complete
        approve Writer
        reject Writer
        ignore Manager
```

Now you want to allow the editor to approve or reject the object. You also want to add a second signature that shows the manual has been accepted by the editor. To make these changes, you could write a Modify Policy statement similar to the following:

```
modify policy "Manual Release" state "In Progress"
    signature Complete
        add approve Editor
        add reject Editor
    signature Accepted
        add approve Editor
        add reject Editor
        add ignore Manager;
```

After this statement is processed, the state definition appears as:

```
state In Progress
    revision false
    version true
    public all, notenable, notdisable, notoverride
    owner all, notenable, notdisable, notoverride
    user Manager override
    signature Complete
        approve Writer, Editor
        reject Writer, Editor
        ignore Manager
    signature Accepted
        approve Editor
        reject Editor
        ignore Manager
```

Deleting a Policy

If you decide that a policy is no longer required, you may delete it by using the Delete Policy statement:

```
delete policy NAME ;
```

NAME is the name of the policy to be deleted. If the name is not found, an error message will result.

When this statement is processed, eMatrix searches the list of policies. If the name is found, that policy is deleted IF there are no objects that use the policy. If there are objects that use the policy, they must be reassigned or deleted before the policy can be removed from the policy list.

For example, you might enter this statement to delete the policy named “Performance and Salary Review:”

```
delete policy "Performance and Salary Review" ;
```


18

Working With Business Wizards

Overview of Business Wizards.....	418
What is a Frame?	418
What is a Widget?.....	419
Runtime Program Environment	420
Using \${} Macros	421
Planning the Wizard.....	421
Creating a Business Wizard	422
Code Clause	422
Description Clause.....	423
Wizard Program Type (External or MQL)	423
File Clause.....	423
Icon Clause.....	423
Needs Business Object Clause	423
Downloadable Clause.....	424
Execute Clause.....	424
Hidden Clause	425
Property Clause	425
Frame Clause	425
Programming Business Wizards.....	436
Strategy for Creating Business Wizards	436
Program Environment Variables.....	437
Loading Complex Widgets.....	440
Using Spaces in Strings.....	440
Using \${} Macros	440
Using Eval Statements	443
MQL Download/Upload Commands	444
Download Command	444
Upload Command.....	444
Upload Command vs. Widget Upload Option.....	445
Running/Testing the Business Wizard	447
.Command Line Interface	447
eMatrix Interface.....	447
Invoking a Wizard from a Program	448
Copying and/or Modifying a Business Wizard Definition.....	450
Copying (Cloning) a Wizard Definition	450
Modifying a Wizard Definition	450
Modifying Wizard Frames	452
Modifying Widgets	452
Deleting a Business Wizard.....	454

Overview of Business Wizards

Business Wizards allow you to create a user interface to simplify repetitive tasks performed by eMatrix users.

A *wizard* is a program which asks the user questions and executes a task based on the information received. It consists of a one or more dialog boxes, called *frames*, which are presented sequentially to the user. Generally, each frame provides some explanation or asks a question, then requires that the user either type a response or make a choice from a list. When all information has been collected, the wizard program performs an action based on the information.

When you create a Business Wizard, you choose the number of frames and define the contents of each frame, customizing the wizard for a specific purpose relating to your database.

Suppose, for example, that a number of users are required to check reports into the database on a weekly basis. A wizard could ensure that reports are named according to a standard report naming convention, prevent the accidental replacement of existing files, and track which users have submitted a report, even sending IconMail or email to the manager who reads the reports to advise that the reports have been checked in.

A wizard is a type of Program object. This means that a wizard can be launched most of the places that a Program can be launched, for example, stand-alone, or as a method. However, due to the graphical nature of wizards, they cannot be executed using MQL.

Wizards are composed of *frames*. Each frame contains one or more *widgets*. Each term is explained in detail below.

What is a Frame?

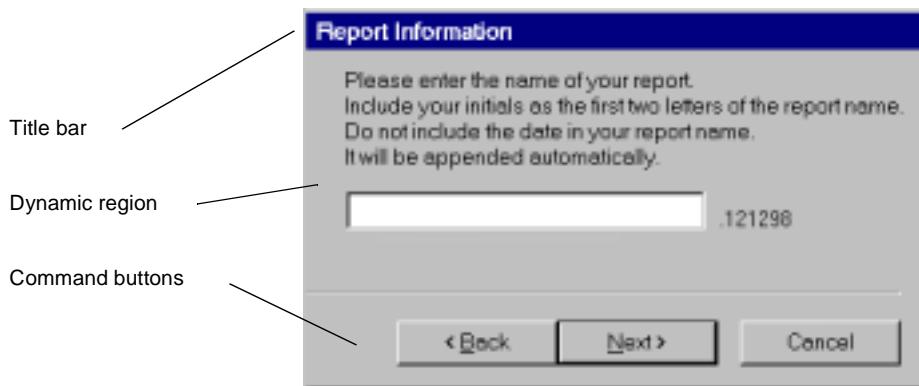
A *frame* is a dialog box that contains instructions and/or asks for information from the user. The information gathered in one frame can be checked for correctness before moving to the next frame and will often dictate the choices loaded into the next frame.

Frames are designed by the Business Administrator using a layout editor. The basic layout of a frame consists of a fixed title in the title bar at the top, a fixed set of three active command buttons along the bottom, and a dynamic region (sub-frame) in the middle. Only the dynamic region can change from one frame to the next.

In order to be effective, a wizard should gather information over a series of simple steps with each step responsible for a single piece of information. This allows for focused activity during each frame and eliminates the need for large complex dialogs.

The frame-specific dynamic region typically has a multiple-line text box near the top that describes the step being performed. The rest of the region is set up to gather the information needed for the step to be completed. The region is called “dynamic” because choices provided in the region can be filled during run time.

The following shows a frame whose dynamic region contains a multiple-line textbox which explains the format for the name of a report. It also provides an input field where the user can type the report name:



For most frames, three command buttons are available: **Back**, **Next**, and **Cancel**. After providing information or making a selection, the user clicks the **Next** button to proceed to the next frame. Clicking the **Back** button returns to the previous frame, where changes can be made to the information supplied. Clicking the **Cancel** button exits the wizard. The first frame of any wizard has a deactivated (grayed-out) **Back** button. The final frame of the sequence has a **Finish** button in the place of the **Next** button. When a user selects the **Finish** button in the final frame, the task is carried out by executing the code associated with the Business Wizard. A single optional status frame may be used to return feedback to the user about the task just performed (or provide the reason for the task not being performed).

What is a Widget?

Anything that is included in the dynamic region of a frame is called a *widget*. The designer of the wizard chooses how many and what types of widgets will be included in each frame.

A frame can include any number of widgets, but it is most effective to limit their numbers in order to reduce the complexity of each frame.

There are ten different types of widgets. The following tables lists the widget types, which are explained in more detail in [Widget Subclause](#).

Widget	Description
label	a non-editable text field
command button	a button which, when clicked, executes a program
text box	a text field, which can be defined to be either editable or non-editable
image	a picture file containing a GIF image
list box	a list of items from which a single or multiple selections can be made
combo box	a combination of an editable text box and a list box—users can type their choice or select from the list

Widget	Description
radio button	a group of one or more mutually exclusive options, each with a circle beside it; only one can be selected
check box	a group of one or more options, each with a check box beside it; multiple options can be selected
directory text box	a text field with an ellipsis button next to it, which launches the directory chooser.
file text box	a text field with an ellipsis button next to it, which launches the file chooser.

Runtime Program Environment

Business Wizards require a mechanism for passing information between the Program objects that can be used within the wizard. The *Runtime Program Environment* (RPE) is provided to hold program environment variables. The RPE, which is a dynamic heap in memory that is used to hold name/value pairs of strings, makes it possible to pass parameters between internal programs and scripts.

For Business Wizards, the RPE allows information to be passed between frames, between programs that control the individual widgets, and to the wizard program that processes the information gathered from the user. For example, after all data-gathering frames have been displayed, you may want to present a summary frame where the user can check for errors. Also, if the user clicks the **Back** button, you would want the frames to display just as they did when the user clicked the **Next** button, including the choices the user made.

See [Runtime Program Environment](#) for additional information on the RPE.

Business Wizard Internal Programs

The basic functions of the Business Wizard require no specific programming since they are handled internally by the system. These basic functions include displaying frames and widgets, including responding to the Next, Back and Cancel buttons.

The Business Administrator customizes each Business Wizard by writing programs that control the individual widgets and that execute a task based on the information collected from the user. The programs that can be used to customize the Business Wizard include the following:

- **Prologue** – the program that executes just before a frame is displayed. See [Prologue Subclause](#).
- **Epilogue** – the program that executes when a frame is closed. See [Epilogue Subclause](#).
- **Load Program** – the program that loads values into the widget field. See [Load Subclause](#).
- **Validate Program** – the program that tests the validity of the widget's state. See [Load Subclause](#).
- **Button Program** – the program that controls what happens when a command button located in the dynamic area of a frame is selected.
- **Wizard Code** – the program that controls the task that is executed when the user clicks the Finish button in the Business Wizard. See [Code Clause](#).

Note that all of the programs listed above (except for the actual wizard code) must be defined as independent program objects before they can be used by a Business Wizard. You can edit existing programs while defining a wizard but you cannot create a new program object while editing a wizard.

Business wizard internal programs should not launch external applications, such as a word processing program. Attempts to do so will cause unexpected results, including crashing the wizard.

Using \${} Macros

Macros provide a relatively easy way to work with variables. The main limitation to macros however is that they are available only for the program that calls them. When working with nested programs, the RPE enables you to pass values between programs. \${} macros are placed into the RPE so that they can be used by nested programs. Just remember to drop the “\${}” characters when referencing the macro variable in the RPE.

Macros that are specific to Business Wizards are shown in the *Macros Appendix*.

Planning the Wizard

Before you start creating a Business Wizard, you should plan what the wizard will accomplish and how you want it to look.

- Decide which task(s) the Business Wizard will perform, for example, checking in a report.
- Determine what information is needed to perform the task, for example, user name, department name, report name, etc.
- Based on the amount of information that needs to be collected, decide how many frames are needed and how each frame should look.
- Decide what platform from which users will be running the Wizard. The wizard should be designed with the lowest screen resolution in mind and the Master Frame should be sized to fit the screen. Also, if the wizard will be executed from the Web, it is recommended that default fonts and colors are used. The `font.properties` file located in the browser's directory specifies what fonts are displayed, but this file is generally used for specifying fonts for alternate character sets for Asian languages. For more information on the `font.properties` file, and setting fonts on the web, see <http://java.sun.com/products/jdk/1.1/docs/guide/intl/fontprop.html>.

Creating a Business Wizard

Creating a Business Wizard involves setting up basic Wizard parameters, defining frames and widgets (the components of frames), and specifying the underlying program code.

Wizards are defined using the Add Wizard statement:

```
add wizard NAME [ ADD_ITEM {ADD_ITEM} ];
```

NAME is the name of the wizard you are creating. All wizards must have a name assigned. When you create a wizard, you should assign a name that has meaning to both you and the user. The wizard name is limited to 127 characters. For additional information, refer to *Business Object Name* in Chapter 34.

ADD_ITEM is an Add Wizard clause which provides additional information about the wizard. The Add Wizard clauses are:

code CODE
description VALUE
external mql
file FILENAME
icon FILENAME
[! not]needsbusinessobject
[! not]downloadable
execute immediate deferred
frame FRAME_NAME FRAME_ITEM {FRAME_ITEM}
[! not]hidden
property NAME [to ADMINTYPE NAME] [value STRING]

All of these clauses are optional. You can define a wizard by simply assigning a name to it. You will learn more about each Add Wizard clause in the sections that follow.

Code Clause

The Code clause of the Add Wizard statement defines the commands for the wizard. The code provides the instructions to act on the information collected from the user during the execution of the frames belonging to the Business Wizard.

Include MQL/Tcl program commands if you specify MQL as the wizard program type.

If you specify external as the wizard program type, include the command necessary to start the external program. Include path information, if necessary. For example:

```
c:\matrix\programs\buswiz3.exe
```

Because Business Wizards are made up of a number of programs, there must be a way to pass information between the programs. The method used is the Runtime Program Environment (RPE), a dynamic heap in memory that is used to hold name/value pairs of strings. See the *Runtime Program Environment* section for additional information.

Description Clause

The Description clause of the Add Wizard statement can provide general information about the purpose of the wizard. Since there may be subtle differences between wizards, the Description clause enables you to point out these differences.

There is no limit to the number of characters you can include in the description. However, keep in mind that the description is displayed when the mouse pointer stops over the wizard in the Program chooser. Although you are not required to provide a description, this information is helpful when a choice is requested.

Wizard Program Type (External or MQL)

You can specify the source of the code as MQL or external. An MQL program can be run from any machine with eMatrix installed; it is not necessary for MQL to be available on the machine.

```
add wizard NAME mql;
```

An external wizard program consists of commands in the command line syntax of the machines from which they will be run. Since MQL can be launched from a command line, MQL code could be specified in an external program. This would spawn a separate MQL session that would run in the background. In this case, MQL would have to be installed on every machine which will run the wizard.

```
add wizard NAME external;
```

File Clause

Specify the file that contains the code for the wizard. The code provides the instructions to act on the information collected from the user during the execution of the frames belonging to the Business Wizard.

This is an alternative to use the Code clause to define commands for the wizard.

Icon Clause

The Icon clause associates a special image with a wizard. When a user searches for a wizard, the icon can help identify the object to select. You can assign a special icon to the new wizard or use the default icon. The default icon is used when in view-by-icon mode. Any special icon you assign is used when in view-by-image mode. When assigning a unique icon, you must use a GIF image file. Refer to [Icon Clause](#) in Chapter 1 for a complete description of the Icon clause.

GIF filenames should not include the @ sign, as that is used internally by eMatrix.

Needs Business Object Clause

You can specify that the wizard must function with a business object. This selection assumes that you will be adding the Business Wizard to a business Type as a method.

For example, you would select this option if the wizard promotes a business object. If, however, the wizard creates a business object, the wizard is independent of an existing object and this option would not apply.

```
add wizard NAME needsbusinessobject;
```

eMatrix would run any wizard specified as “needs business object” with the selected object as the starting point. If a wizard does not require a business object, the selected object would not be affected. The following indicates that a business object is not needed:

```
add wizard NAME !needsbusinessobject;
```

When defining a type or format, you can specify program information:

Downloadable Clause

If the wizard includes code for operations that are not supported on the Web product (for example, Tk dialogs or reads/writes to a local file) you can include the `downloadable` clause. If this is included, this program is downloaded to the web client for execution (as opposed to running on the Collaboration Server). For wizards not run on the Web product, this flag has no meaning.

```
add wizard NAME downloadable;
```

If the `downloadable` clause is not used, `notdownloadable` is assumed.

If the `downloadable` clause is used, then `deferred` program execution is assumed (see the [Execute Clause](#)). If the `downloadable` clause is used, and the `execute` clause is `immediate`, an error will be generated that reads:

A program that is downloaded cannot execute immediately.

Execute Clause

Use the `execute` clause to specify when the wizard should be executed. If the `execute` clause is not used, `immediate` is assumed.

`Immediate` execution means that the program runs within the current transaction, and therefore can influence the success or failure of the transaction, and that all the program’s database updates are subject to the outcome of the transaction.

`Deferred` execution means that the program is cued up to begin execution only after the outer-most transaction is successfully committed. A deferred program will not execute at all if the outer transaction is aborted. A deferred program failure only affects the new isolated transaction in which it is run (the original transaction from which the program was launched will have already been successfully committed).

However, there are a number of cases where deferring execution of a program does not make sense (like when it is used as a trigger check, for example). In these cases the system will execute the program immediately, rather than deferring it until the transaction is committed.

There are four cases where a program’s execution can be deferred:

- Stand-alone program
- Method
- Trigger action
- State action

There are six cases where deferred execution will be ignored:

- Trigger check
- Trigger override
- State check
- Range program

- Wizard frame prologue/epilogue
- Wizard widget load/validate

There is one case where a program's execution is always deferred:

- Format edit/view/print

A program downloaded to the web client for local execution (see [Downloadable Clause](#)) can be run only in a deferred mode. Therefore, if you use the downloadable option, the program is automatically deferred.

Hidden Clause

You can specify that the new wizard is “hidden” so it does not appear in the Program or Methods choosers in eMatrix, which simplifies the end-user interface. A wizard can contain a number of programs which are not intended to be executed as stand-alone programs (such as the load and validate programs for widgets) and users should not be able to view these program names in the eMatrix Program chooser. Users who are aware of a hidden program’s existence can enter its name manually where appropriate. Hidden objects are also accessible through MQL.

Property Clause

Integrators can assign ad hoc attributes, called Properties, to the wizard. Properties allow associations to exist between administrative definitions that aren’t already associated. The property information can include a name, an arbitrary string value, and a reference to another administration object. The property name is always required. The value string and object reference are both optional. The property name can be reused for different object references, that is, the name joined with the object reference must be unique for any object that has properties.

```
add wizard NAME  
    property NAME [to ADMINTYPE NAME] [value STRING];
```

For additional information on properties, see *Overview of Administration Properties* in Chapter 24.

Frame Clause

A Business Wizard is composed of one or more *frames*. Each frame is a window that requests and stores information in order to complete a task. You can include any amount of information in a frame, but in order to be effective, each frame should be responsible for gathering a single piece of information, for example, a department name or the name of an object in the database.

The Frame clause of the Add Wizard statement defines all information related to a wizard frame including: units and color of the frame, prologue and epilogue programs, status, and widget types. The Frame clause uses the following syntax:

```
frame FRAME_NAME [ FRAME_ITEM { FRAME_ITEM } ]
```

FRAME_NAME is the name of the frame you are defining. All frames must have a name assigned. This name must be unique within the wizard and should have meaning for both you and the user. The frame name is limited to 127 characters. For additional information,

refer to *Business Object Name* in Chapter 34. The following are some examples of names you might use:

Department Name
Get Codeword
Get Filename
Availability Choices

FRAME_ITEM is a Frame subclause which provides additional information about the frame. The Frame definition subclauses are:

units [picas points inches]
color FOREGROUND [on BACKGROUND]
size WIDTH HEIGHT
prologue PROG_NAME [input ARG_STRING]
epilogue PROG_NAME [input ARG_STRING]
status [true false]
icon FILENAME
widget WIDGET_TYPE WIDGET_ITEM {WIDGET_ITEM}

The sections that follow describe these clauses and the arguments they use.

Units Subclause

The Units subclause of the Frame clause specifies the units of frame measurement. There are three possible values: picas, points, or inches.

units picas <i>Or:</i> units points <i>Or:</i> units inches

eMatrix will automatically assume a picas value if you do not use a Units clause.

Picas are the most common units of page measurement in the computer industry. Picas use a fixed size for all characters. Determining the size of a field value is easy when using picas as the measurement unit. Simply determine the maximum number of characters that will be used to contain the largest field value. Use that value as your field size. For example, if the largest field value will be a six digit number, you need a field size of six picas. This is not true when using points.

Points are standard units used in the graphics and printing industry. A point is equal to 1/72 of an inch or 72 points to the inch. Points are commonly associated with fonts whose print size and spacing varies from character to character. Unless you are accustomed to working with points, measuring with points can be confusing and complicated. For example, the character "I" may not occupy the same amount of space as the characters "E" or "O." To determine the maximum field size, you need to know the maximum number of characters that will be used and the maximum amount of space required to express the largest character. Multiply these two numbers to determine your field size value.

Inches are common English units of measurement. While you can use inches as your unit of measurement, be aware that field placement can be difficult to determine and specify. Each field is composed of character string values. How many inches does each character need or use? If the value is a four-digit number, how many inches wide must the field be to contain the value? How many of these fields can you fit on a frame? Considering the problems involved in answering these questions, you can see why picas are a favorite measuring unit.

When planning the wizard, consider the operating systems of those who will use the wizard. The wizard should be designed with the lowest screen resolution in mind and the Master Frame should be sized to fit the screen.

Color Subclause

The Color subclause of the Frame clause specifies color values used as the default foreground and background for the frame.

```
color [FOREGROUND] [on BACKGROUND]
```

foreground is the name of the color for the foreground printed information (any vertical or horizontal lines of information).

background is the name of the color used as an overall background for the frame. Note that the word `on` is required only if a background color is specified.

For a list of available colors, refer to:

- Windows— `\$MATRIXHOME\lib\winnt\rgb.txt`.
- UNIX— `/\$MATRIXHOME/lib/ARCH/rgb.txt`.
\$MATRIXHOME is where the eMatrix application is installed.
ARCH is the UNIX platform.

Size Subclause

The Size subclause of the Frame clause defines the dimensions of the frame. A frame can be any size.

To define a frame size, you need two numeric values. One represents the width (COL_SIZE) and one represents the length (ROW_SIZE). Both of these values must be provided and entered according to the following syntax:

```
size ROW_SIZE COL_SIZE
```

The values reflect the Units value defined for the frame (picas, points, or inches). For example, for an 7 by 5 inch frame, the following are equal:

<code>size 66 30</code>	Measured in picas.
<code>size 504 360</code>	Measured in points.
<code>size 7 5</code>	Measured in inches.

When selecting a frame size, keep in mind the monitor sizes of the users who will be running the wizard. For example, a wizard frame does not display the same amount of information on a screen with a resolution of 640x480 as it does on a screen with a resolution of 1280x1024. Use the lowest common denominator of screen resolution to design wizards and the master frame should be fit to size.

Prologue Subclause

The frames's *prologue* is a program which executes immediately before a frame is displayed and before any other action is taken. Prologue programs can be used to influence the loading of widgets. They can also be used to skip frames. Each frame within a Business Wizard can have its own prologue.

The prologue program can be used to prepare for the loading of the widgets. This might include setting values in the RPE, and even redefining widget load programs. This allows widget load functions to be generic, with specific behavior being controlled at run time by the owning frame.

The prologue function can also cause the frame to be skipped. Whenever a prologue program returns a non-zero value, the frame is skipped.

The following syntax is used:

```
prologue PROG_NAME [input ARG_STRING]
```

ARG_STRING defines arguments to be passed into the program. The arguments can be referenced by variables within the program. Variables 0, 1, 2... etc. are reserved by the system for passing in arguments.

- Environment variable "0" always holds the program name and is set automatically by the system.
- Arguments from ARG_STRING are set in environment variables "1", "2",... etc.

See [Runtime Program Environment](#) for additional information on program environment variables.

One common use of the prologue program is to skip a frame within a wizard. If a frame's prologue exits with a non-zero return code, that frame will be skipped. This works either going forward or backward within a wizard. In the following example, the user is looking at Frame C and tries to click the **Back** button. You don't want the user to be able to go back to Frame B until certain information has been filled in on Frame C.



To accomplish this, you could include the following code in the prologue for all frames before Frame C (in this case, Frame B and Frame A, since you don't want to go back to either).

```
tcl;
eval {
set sFrameMotion [string toupper [ mql get env FRAMEMOTION ] ]
if { $sFrameMotion == "BACK" } {
    set iExitCode 1
} else {
    set iExitCode 0
}
exit $iExitCode
}
```

Since a frame's prologue is executed before the frame displays, if it returns an exit code of 1, the frame is skipped. In this case, the user would continue to see Frame C.

Epilogue Subclause

The frame's *epilogue* is a program which is executed whenever a frame is closed. The epilogue program can be used to undo what the prologue program did, and perform any other cleanup activity. Since data is passed between frames using RPE, you can use the epilogue program to clean up widget variables before moving on to the next frame.

If an epilogue program returns non-zero after pressing the **Next** button, the current frame is redisplayed. Thus, even if all of the widget validate programs return zero, the epilogue program can still keep the next frame from appearing. Although the epilogue program runs when the **Back** button is pressed, the return code is not checked.

When the epilogue program for the last frame in the frame sequence has been executed, programmatic control of the wizard is then handled by the wizard code, defined on the Code tab of the Business Wizard. See [Code Clause](#) for additional information.

The following syntax is used:

```
epilogue PROG_NAME [ input ARG_STRING ]
```

ARG_STRING defines arguments to be passed into the program. The arguments are referenced by variables within the program. Variables 0, 1, 2... etc. are reserved by the system for passing in arguments.

- Environment variable “0” always holds the program name and is set automatically by the system.
- Arguments from ARG_STRING are set in environment variables “1”, “2”,... etc.

See [Runtime Program Environment](#) for additional information on program environment variables.

Status Subclause

Each wizard can have an optional *status frame*, which is generated after the user has clicked the Finish button and the Business Wizard code has been executed. This option can be applied to the last frame of the sequence only.

The status frame returns feedback to the user about the task just performed or provides the reason why the task was not performed. The status frame contains a single **Close** button in place of the three regular frame buttons (Back, Next, Cancel). The dynamic region would typically contain only a multiline text box, though it could also contain image or label widgets.

Status frame processing includes executing the prologue and the load programs, but no input from the user is accepted or processed. The status frame programs should not perform any database operations.

The **status** subclause uses two arguments: true and false:

```
status [true|false]
```

When the status argument is set to **true**, the frame is used as a status frame. If the status argument is set to **false** (this is the default), no status frame is generated at the conclusion of the Business Wizard.

Icon Subclause

The Icon subclause of the Frame clause associates a special image with a frame. When a user searches for a frame, the icon can help identify the frame to select. Refer to [Icon Clause](#) in Chapter 1 for a complete description of the Icon clause.

Widget Subclause

The term *widget* refers to any component of the frame. The Widget subclause of the Frame clause uses the following syntax:

```
widget WIDGET_TYPE WIDGET_ITEM {WIDGET_ITEM}
```

WIDGET_TYPE refers to one of ten types of fields that make up the dynamic region of the frame. Widget types include the following:

Widget Type	Meaning
label	a non-editable text field.
button	a button which launches a program object.
textbox	a text field which can be defined to be either editable or non-editable.
image	a picture file containing a GIF image.
listbox	a box containing a list of items from which multiple selections can be made.
combobox	a box containing a list of items from which a single selection can be made.
radiobutton	a group of one or more options, each with a circle next to it. A blank circle indicates that it is not selected (or "off"); a filled in circle indicates that it is selected (or "on"). A radio group is used for mutually exclusive choices, that is, only one option can be selected.
checkbox	a group of one or more options, each with a check box next to it. A blank check box indicates that it is not selected (or "off"); a box with a check in it indicates that it is selected (or "on"). Multiple options can be selected.
directorytextbox	a text field with an ellipsis button next to it, which launches the directory chooser.
filetextbox	a text field with an ellipsis button next to it, which launches the file chooser.

WIDGET_ITEM is a Widget subclause which provides additional information about the widget. The Widget subclauses are:

```
name WIDGET_NAME  
value VALUE [selected value|input ARG_STRING]  
load PROG_NAME [input ARG_STRING]  
validate PROG_NAME [input ARG_STRING]  
observer PROG_NAME [input ARG_STRING]
```

color FOREGROUND [on BACKGROUND]
start XSTART YSTART
size WIDTH HEIGHT
font FONT_NAME
autoheight [false true]
autowidth [false true]
drawborder [false true]
multiline [false true]
edit [false true]
scroll [false true]
password [false true]
upload [false true]

The widget subclauses are described in the sections that follow.

Widget Name Subclause

The Name subclause is used to define a unique name for the widget. Widget names can be any length of alphanumeric characters; spaces can be included.

Since widgets are named so that they can be identified by the system, a unique name is automatically created by the system for each widget. Therefore, this subclause is optional. If you do not specify a name for the widget, the system-supplied name is used.

If you want to include the widget on multiple frames within the Business Wizard and retain the value within each frame, you should replace the system-supplied name with a name of your own choosing. Widgets that share the same name will share the same value.

The widget Name subclause has the following syntax:

widget WIDGET_TYPE name WIDGET_NAME

Value Subclause

The Value subclause of the Widget subclause identifies the contents of the widget. The Value subclause has the following syntax:

value VALUE [selected value input ARG_STRING]

For all widgets except the image and command button widget types, VALUE specifies the text that will be included in the widget, for example, the string of characters to be included as a label. For textbox or listbox widgets, VALUE is the default text that is included in the box. For combobox, checkbox, or radiobutton, VALUE is the text included in the group.

Several widgets need two pieces of information: a list of choices and one or more selections. Selected value is an optional argument which identifies the item you want selected or highlighted. For example, if you have a radio group that contains the days of the week and you want “Wednesday” to be selected by default, use the following syntax:

value Monday Tuesday Wednesday Thursday Friday selected value Wednesday

For the image widget type, VALUE is the name of the .gif file used to represent the image. For example:

```
value hands.gif
```

For the command button widget type, VALUE is the name of the program file used to represent the graphic. ARG_STRING is an optional argument(s) that can be passed into the program.

Load Subclause

The load program, if defined, contains code to load values into the widget field. Load is an option for all widgets except for label and image. The load program can also be used to define alternate values for the widget depending on the context, information gathered in a prior frame, etc. The following syntax is used:

```
load PROG_NAME [input ARG_STRING]
```

ARG_STRING defines arguments to be passed into the program. The arguments are referenced by variables within the program. Variables 0, 1, 2... etc. are reserved by the system for passing in arguments.

- Environment variable “0” always holds the program name and is set automatically by the system.
- Environment variable “1” always holds the widget name and is also set automatically by the system.
- Arguments from the Input field are set in environment variables “2”, “3”,... etc.

If a command button program returns non-zero, the frame is refreshed. This means that all widgets in the frame will have their load program run. This allows a command button program to modify variables in the RPE and then force the widgets in the current frame to reload themselves.

Validate Subclause

Validate programs check if the values added by the user (to an input field, for example) are within an acceptable range of parameters. Validate is an option for all widgets except for label and image.

The validate program, if defined, is used to test the validity of the widget’s state. By returning a non-zero value, the validate program causes the system to redisplay the frame and place focus on the offending widget. It is good style to have the validate program additionally display a message box that explains the error.

Validate programs are normally used for editable text box widgets and combo box widgets.

Note that the validate program does not execute when the user clicks the Back button.

The following syntax is used:

```
validate PROG_NAME [input ARG_STRING]
```

ARG_STRING defines arguments to be passed into the program. The arguments are referenced by variables within the program. Variables 0, 1, 2... etc. are reserved by the system for passing in arguments.

- Environment variable “0” always holds the program name and is set automatically by the system.

- Environment variable “1” always holds the widget name and is also set automatically by the system.
- Arguments from the Input field are set in environment variables “2”, “3”,... etc.

Observer Subclause

The Observer subclause of the Widget subclause defines a program to be executed when the user makes a selection from a wizard list box, check box, or radio button.

The following syntax is used:

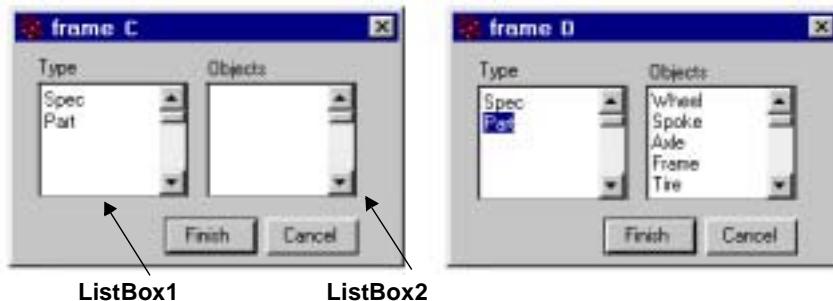
```
observer PROG_NAME [input ARG_STRING]
```

ARG_STRING defines arguments to be passed into the program. The arguments are referenced by variables within the program. Variables 0, 1, 2... etc. are reserved by the system for passing in arguments.

- Environment variable “0” always holds the program name and is set automatically by the system.
- Arguments from ARG_STRING are set in environment variables “1”, “2”,... etc.

See [Runtime Program Environment](#) for additional information on program environment variables.

For example, the selection of Type “Part” from the ListBox1 calls an observer program that populates the Business Objects in ListBox2, as illustrated below.



The observer program has all the side effects of clicking a button within a wizard; that is, it is executed immediately and if it exits with a return code of 1, each of the widgets in the current frame has its values (re)read and (re)populated by the RPE variables associated with the frame. The INVOCATION macro is populated with the value “observer.” If the widget was a multi-selection list box, then `mql get env <widget name>` returns a space-delimited string containing current values.

Performance is an important consideration. When calling an observer program, every widget on a frame is examined to get the current value of that widget. For a frame with many widgets, this could be time consuming. Also, since the observer program is actually run on the server, there is lag time associated with the round trip from client to server and back. The developer of an observer program needs to evaluate whether the time it takes for the observer program to complete is acceptable for a user to wait.

Color Subclause

The Color subclause of the Widget subclause specifies color values used as the default foreground and background for the widget.

```
color [FOREGROUND] [on BACKGROUND]
```

FOREGROUND is the name of the color for the foreground printed information (any vertical or horizontal lines of information).

BACKGROUND is the name of the color used as an overall background for the widget. Note that the word on is required only if a background color is specified.

A list of available colors is contained on Windows in \\$MATRIXHOME\lib\winnt\rgb.txt.

Start Subclause

The Start subclause of the Widget subclause is used to define the placement of the widget in the frame. The syntax is:

```
widget WIDGET_TYPE start XSTART YSTART
```

The XSTART (horizontal) and YSTART (vertical) coordinates identify the widget's starting point—where the first character of the field value is displayed. For example, to place a label widget in the upper left corner of the frame, the starting position would be 0, 0:

```
widget label start 0 0
```

Size Subclause

The Size subclause of the Widget subclause is used to define the size of the widget. Specify height and width values to define the widget size. The syntax is:

```
widget WIDGET_TYPE size WIDTH HEIGHT
```

The widget's size is dependent on the Units (picas, points or inches) specified for the widget. The width value defines the horizontal size of the widget. The height value defines the vertical size of the widget. For example, to define a textbox widget 12 picas long by 3 picas high, use:

```
widget textbox units picas size 12 3
```

Font Subclause

Use the Font subclause to specify the font in which the text of a widget field appears. This option is available for all widget types except for the image widget. The syntax is:

```
widget WIDGET_TYPE font FONT_NAME
```

Fonts available are based on the computer's defined system fonts.

Although it is possible to change the fonts used in wizards, it is best to let a wizard use the default font and colors which have been set up by the end user, particularly if they will be used across the Web (for example, in using wizards with Info Central). See the discussion *Wizard Fonts on the Web* in the *Business Modeler Guide*.

Autowidth and Autoheight Subclauses

Specify true or false for the Autowidth and Autoheight subclauses of the Widget subclause if you want the program to select the appropriate height and/or width based on the contents of the widget. The syntax is:

```
widget WIDGET_TYPE autoheight [true\false] autowidth [true|false]
```

Drawborder Subclause

Use the `drawborder` subclause to include a border around the field. This option is available for the text box, image, check box, radio button, file text box, and directory text box widget types. The syntax is:

```
widget WIDGET_TYPE drawborder [false|true]
```

Multiline Subclause

Use the `multiline` subclause to have text displayed on more than one line in a text box or to allow more than one choice to be selected in a list box. The syntax is:

```
widget WIDGET_TYPE multiline [false|true]
```

Edit Subclause

Use the `edit` subclause to allow the user to change the value while viewing the frame. This option is available only for the text box widget type. The syntax is:

```
widget WIDGET_TYPE edit [false|true]
```

Scroll Subclause

Use the `scroll` subclause to have scroll bars displayed so that the user can scroll text up and down. This option is available only for the text box widget type. The syntax is:

```
widget WIDGET_TYPE scroll [false|true]
```

Password Subclause

Use the `password` subclause to have all text the user types into the field masked with the asterisk character. This option is available only for the text box widget type. The syntax is:

```
widget WIDGET_TYPE password [false|true]
```

Upload Subclause

Use the `upload` subclause to allow the user to select a client-side file and copy it to the server. This option is available only for the file text box widget type. See [Upload Command vs. Widget Upload Option](#) for additional information. The syntax is:

```
widget WIDGET_TYPE upload [false|true]
```

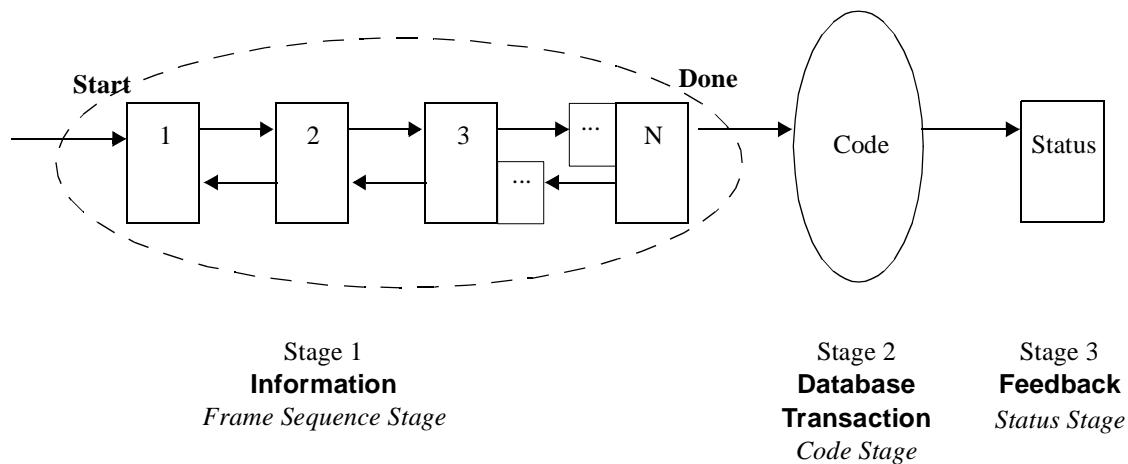
Programming Business Wizards

In creating a Business wizard, you will need to write at least one program, which will perform the database transaction(s) for which the wizard is created. You can also write programs to control the loading of frames and widgets and to check the validity of widget values.

Before you begin to create a Business wizard, you should be familiar with the information contained in *Overview of Programs* in Chapter 19, including information about the Runtime Program Environment (RPE).

As you plan the project, keep in mind that a wizard has three stages:

- **Stage 1** — Information. This stage consists of a series of frames, which gather information from the user. The information is stored in the RPE, which is used to pass data between frames. **No database updates should be performed during this stage.** This is important so that the **Cancel** function can work properly.
- **Stage 2** — Database transaction. The wizard code, defined in the New Wizard dialog box Code tab, is executed. This stage uses the information from Stage 1 to perform all necessary database transactions. Results can be placed in the RPE for use by the optional Status Frame.
- **Stage 3** — Feedback. A single status frame is displayed to inform the user of the status of the transaction performed by the wizard. This stage is optional. It is important to note that this stage follows stage 2 and is not part of the sequence of frames in Stage 1.



Strategy for Creating Business Wizards

The following is one strategy that can be used in creating a Business wizard. The order of the steps can, of course, be changed according to your own programming style.

1. **Plan the project.** Decide what questions you want to ask the user and what format they will use to supply answers. Will they type in their answers, or choose from a list or group? Input fields provide the most flexibility, but it is easier to control the user's choices and prevent errors by having them make a selection from a fixed number of items.

- 2. Design the wizard.** Decide the number of frames and the layout of the widgets in each frame. It is most effective to have each frame request a single piece of information.

The frame sequence should only be used for the task of gathering information from the user, deferring any database work until the internal code of the wizard is executed.
- 3. Design the master frame.** Master frames allow you to easily create a polished look for the Business wizard by making each frame the same size with the same color scheme. You can also add widgets that you want to appear in each frame, for example a logo or a horizontal rule. These default attributes can be overridden, as needed, on a frame-by-frame basis.
- 4. Create the wizard**, positioning widgets on the frames. As you add frames, they inherit the characteristics of the master frame. If, however, you subsequently make changes to the colors or the dimensions of the master frame, these changes will *not* be reflected in any frames already added. Therefore, be sure to set up the master frame before adding any new frames.
- 5. Write code to load and validate widgets**, if the widgets require load or validate programs.
- 6. Write code for prologue and epilogue programs**, if frames require prologue or epilogue programs.
- 7. Add program names** (for load, validate, prologue, epilogue) on the Edit Widget and Edit Frame dialog boxes of the Business wizard you created.
- 8. Write code to execute the task** based on the information collected in the wizard. Include the code on the New Wizard dialog box of the Business wizard.

Program Environment Variables

Because business wizards are made up of a number of independent programs that may need to share data, there must be a way to pass information between the programs. The method used is the Runtime Program Environment (RPE), a dynamic heap in memory that is used to hold name/value pairs of strings.

In using the RPE to pass data between program objects, careful attention must be paid to the naming of program environment variables and to the cleaning up of these variables. All data is passed in the form of strings. Here are some guidelines:

- The program name is automatically associated with a variable named “0”.
- The widget name is automatically associated with a variable named “1”.
- Program input parameters are named “1”, “2”, etc. except within a load or validate program for a widget. Since the variable “1” is reserved for the widget name, program input parameters are named starting with the number “2”.
- The calling program can manually place the expected local variables into the RPE (using the naming scheme above) before executing the called program, or simply add them to the end of the execute command; the system will automatically place them in the RPE with the proper names.
- Programs should return their data in a variable with the name identified with the 1st input parameter.
- Lists should be returned using a Tcl form (space separated, with curly braces used to surround items that contain spaces or special characters).

- Since the RPE is shared memory, any variable can be overwritten by a variable of the same name. It is good practice to always capture wizard variables immediately into custom local variables to be assured that the values will be available where necessary within the wizard. RPE variables can be saved as local variables (in Tcl: `set localUser [mql get env USER]`) or saved back into the RPE under a different name that won't be overwritten (for example: `programName.USER` where `programName` is the name of the program).

Any eMatrix program or trigger that runs at any time before the wizard completes has the potential to alter RPE variables. For example, a user starts a wizard, then checks the attributes of another object in the database. Depending on how the object is configured, a program or trigger may fire automatically without the user's knowledge that could overwrite existing RPE variables.

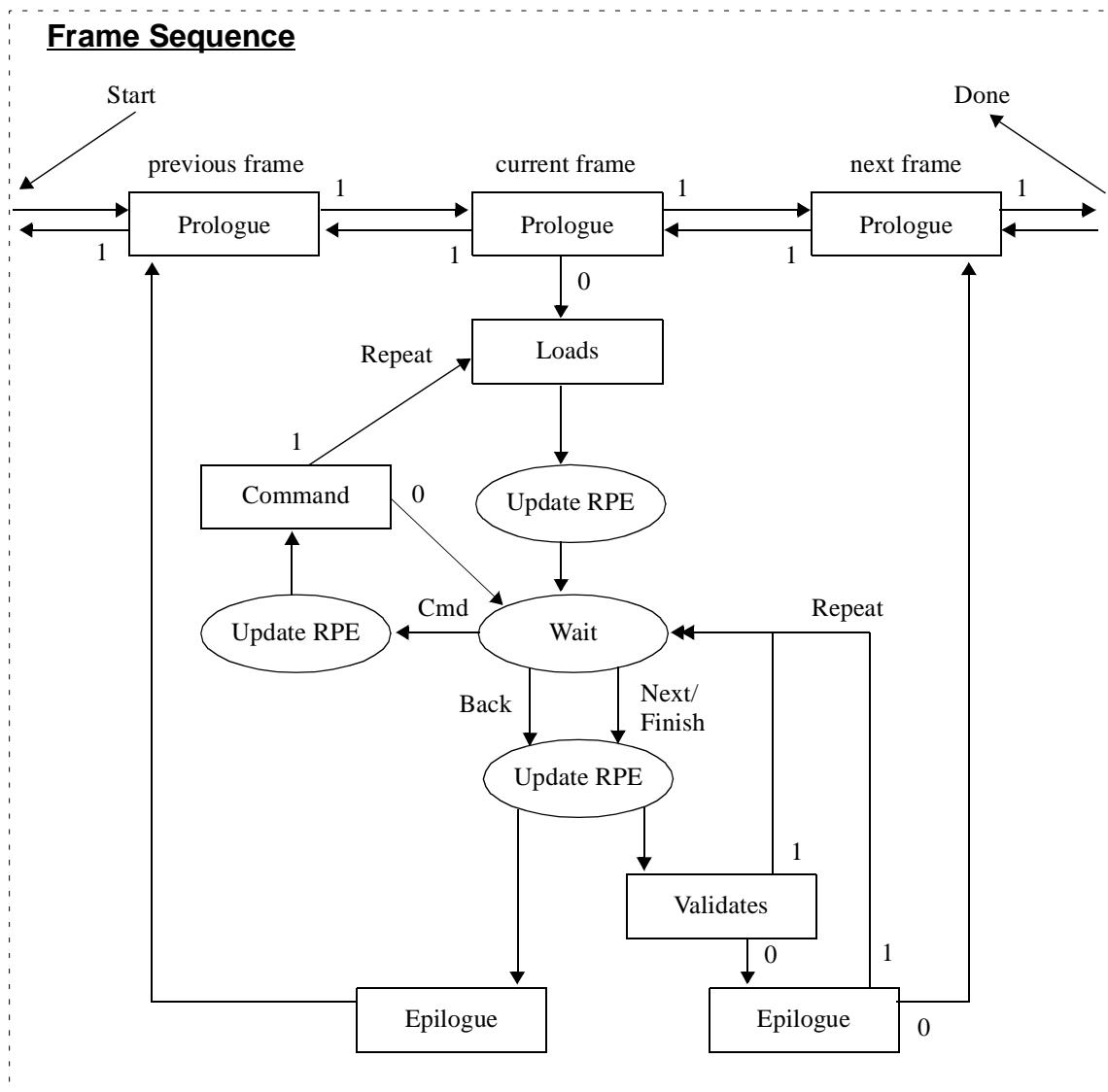
For this reason, saving the Type, Name, Revision, and Objectid into wizard-specific RPE variables should always be done in the prologue of the first frame. This is the only way to preserve this information. A properly written wizard/program should *always* fetch out of the RPE all variables needed before proceeding.

How the System Displays Frames and Widgets

With the previous naming conventions in mind, here is how the frame and widget functions work:

1. When a frame is displayed, its prologue function is first executed. The prologue should establish any parameters in the RPE that will be used by the load functions of the widgets belonging to the frame. This allows widget load functions to be generic, with specific behavior being controlled at run time by the owning frame. The prologue function can also cause the frame to be skipped by returning a non-zero value.
2. Each widget belonging to the frame will then be constructed and set using the following steps:
 - a) If an RPE variable exists with the widget's name, the system uses its value. The variable may already exist if a previous frame has a widget of the same name, or if the frame's prologue program created the variable.
 - b) Assuming no RPE variable is found, the system next checks to see if the widget has a load program. If so, the load program is executed, and again a search is made for an RPE variable with the widget's name. (Each load program is given the widget's name in argument 1.)
 - c) Assuming no RPE variable is found, the system uses the widget's default value found in the value fields on the Edit Widget dialog box.
3. All widgets, except Label and Graphic, have an RPE variable that has the widget's name and has a value equal to its current state. For example, a widget named `answer` is a radio group made up of the choices Yes and No. This is represented in the RPE with a variable named `answer`; its value is either Yes or No depending on the widget's state. As the state of the widget changes, the value of the widget variable changes.

4. If the **Back** button is pressed, the current frame's epilogue function is called (performing any necessary cleanup), and steps 1 through 3 are repeated for the previous frame.
5. If the **Cancel** button is pressed, the Business wizard immediately ends.
6. If the **Next** or **Finish** button is pressed, each widget (except Labels and Graphics) belonging to the frame executes its validate function. After each save function is executed, the exit code is checked and if it is non-zero, the frame remains displayed and focus is placed on the offending widget. This should happen only with an editable text box or combo box widget since all other widgets should provide the user only with legal choices. The current frame's epilogue function is called, and if there is a next frame, steps 1 through 3 are repeated for that frame.
7. If the **Finish** button is pressed, then the body of the Business wizard code is executed. If a status frame is defined, it is displayed after the code is executed and only steps 1 and 2 are performed (the user will need to press the **Close** button to remove the frame from the screen –essentially step 5).



For each widget type there is a defined format for the widget variable that holds state information. These formats are as follows:

Widget types	Widget variable values
Text box	The actual text (including newlines)
List box	
Combo box	
Radio button	List of selections
Check box	

Loading Complex Widgets

Several widgets need two pieces of information: a list of choices, and one or more selections from the list. For example you might have a check box group with the choices Weekdays, Saturday, and Sunday, and want the choice Weekdays to be selected by default.

Choices and selections can be defined on the Edit Widget dialog box in the fields **Choices** and **Selected**. List choices (and selections) in a single string with a space used as a separator.

When using MQL, set the widget's stored database value to the following:

```
VALUE Weekdays Saturday Sunday SELECTED Weekdays
```

As it reads from the database, the system will load choices into the widget until it finds selected. It then assumes that the tokens that follow are default selections.

Another approach is to use a load program. By convention, the choices go into an RPE variable whose name is the same as the load program (obtained through argument 0) and the selections go into an RPE variable whose name is the same as the widget name (obtained through argument 1). Remember that the widget variable in the RPE holds its state, and the state of a complex widget is its current selections. So the load program has identified the state of the widget by loading the variable associated with the argument 1.

Using Spaces in Strings

Lists of widget choices and selections are given in a single string with spaces used as separators. This means that any choice containing spaces should be quoted.

However, due to the extensive use of the Tcl language for manipulating lists, the Tcl notation for lists is recognized by the system. Therefore curly braces can be used to quote single list items. This is true for all arguments being passed into program objects and all values returned by program objects. The system also does a better job of supporting nested use of curly braces than it does with quote characters.

Using \${} Macros

For compatibility, \${} macros continue to be supported. \${} macros are placed into the RPE so that they can be used by nested programs. Just remember to drop the "\${}" characters when referencing the macro variable in the RPE.

The following macros are used in wizards:

Macro	Meaning
WIZARDNAME	Name of the Business wizard.
FRAMENAME	Name of the current frame.
FRAMENUMBER	Number of the current frame in the frame sequence (excluding master and status frames). Frame numbers begin with 1. The status frame returns a value of 0.
FRAMETOTAL	Total number of frames in the frame sequence (excluding master and status frames).

Macro	Meaning
FRAMEMOTION	<p>Current state of wizard processing:</p> <p>start The wizard has been started; the user has not yet clicked a command button.</p> <p>back The user clicked the Back command button.</p> <p>next The user clicked the Next command button.</p> <p>finish The user clicked the Finish command button.</p> <p>repeat The current frame has been redisplayed.</p> <p>status The current frame is the status frame of the wizard.</p>
SELECTEDOBJECTS	<p>List of currently selected objects to be passed to any program. This macro is populated whenever a program or method is invoked from the toolbar, tool menu (Web Navigator only), right-mouse menu or Methods dialog. The macro is also populated when certain dialogs are launched via the appl command (appl icons, details, indented, star and indentedtable), as long as a dismiss program is defined. (See <i>Application Command</i> in Chapter 19). The macro's value consists of a single string of space-delimited business object IDs for the objects that are selected at the time the program or method is invoked. (For a method, this macro is redundant—it is equivalent to the OBJECTID macro—but it is populated nevertheless for consistency.).</p> <p>The SELECTEDOBJECTS macro can be read into a Tcl string or list variable as follows:</p> <pre># this reads the macro as a single Tcl string set sObjs [mql get env SELECTEDOBJECTS] # this reads the macro into a Tcl list. set env lObjs [split [mql get env SELECTEDOBJECTS]]</pre> <p>When no objects are selected, the macro is created, but holds an empty string.</p>
WIZARDARG	<p>A single RPE variable that is created from the ARG string in an appl wizard command, which can be read by any of the wizard's supporting code to receive input from its 'caller.'</p> <p>If you want to pass multiple pieces of information from the caller to the wizard, you would need to format the information into a single string with an identifiable delimiter (e.g.,). For example, to pass a businessobject type, current state and owner, you could collect them into a Tcl variable \$sArgString as follows:</p> <pre>set sArgString "\$sType \$sState \$sOwner" appl wizard bus \$sBusId MyWizard \$sArgString</pre> <p>Then, the wizard program that reads the WIZARDARG RPE variable could split it up again as follows:</p> <pre>set sArgString [mql get env WIZARDARG] set lArgList [split \$sArgString] set sType [lindex \$lArgList 0] set sState [lindex \$lArgList 1] set sOwner [lindex \$lArglist 2]</pre>

Macro	Meaning
WORKSPACEPATH	Directory that is used as the base directory for upload/download. For programs/wizards run through the server, it is evaluated by combining the ematrix.ini settings MX_BOS_ROOT and MX_BOS_WORKSPACE, and appending a unique temporary directory for the session that runs it to avoid collisions across sessions. For desktop client configurations, it is set to the matrix.ini setting TMPDIR. These settings guarantee that the client has access to the files on the server.

Although the operating system determines the valid syntax of the commands, the typical syntax used is:

```
command ${MACRO 1} ${MACRO 2} ...
```

Using Eval Statements

You can wrap the code in an eval statement to prevent the eMatrix output window from popping up. Placing tcl code in an eval statement causes output to be suppressed when executed within eMatrix.

MQL Download/Upload Commands

The download and upload MQL commands are available for use in programs to be executed by a Web client through a Collaboration Server. They allow client-side files and directories to be manipulated by server-side programs (and vice versa). Generally the download and upload commands would be used in conjunction with a wizard ‘file text box’ widget. See [Widget Subclause](#) for details.

Download Command

Use the download command to download files to the client that are created by wizards on the server, or when a file resides on a server that must be processed by a client-side program.

```
download sourcefile SOURCE targetfile TARGET [[!|not]delete] [[!|not]overwrite];
```

SOURCE specifies the directory path and filename of the file to be downloaded from the server, relative to WORKSPACEPATH. Refer to [Using \\${} Macros](#) for more information on WORKSPACEPATH. Note that when used in a desktop environment, the file is moved from one location to another on the same machine.

TARGET specifies the full path and filename on the client. If the specified directories do not exist, eMatrix will create them.

When delete is specified, the source file is deleted on the server after the download. The default is TRUE.

When overwrite is specified, if TARGET filename already exists, the downloaded file replaces it. The default is FALSE.

For example:

```
download sourcefile report.doc targetfile reports/report.doc !delete overwrite;
```

Upload Command

Use the upload command when a file resides on a client and must be processed by a server side program. Refer [Upload Command vs. Widget Upload Option](#) for more information.

```
upload sourcefile SOURCE targetfile TARGET [[!|not]delete] [[!|not]overwrite];
```

SOURCE specifies the full path and name of the file to be uploaded from the client.

TARGET specifies the destination path and file name on the server, relative to WORKSPACEPATH. If the specified directories do not exist, eMatrix will create them. Refer to [Using \\${} Macros](#) for more information on WORKSPACEPATH .

When delete is specified, the source file is deleted on the client after the upload. The default is FALSE.

When overwrite is specified, if TARGET file name already exists, the uploaded file replaces it. The default is TRUE.

Note that the defaults for the delete and overwrite options are different between upload and download in order to maintain consistency with the default behavior of checkin/checkout.

For example:

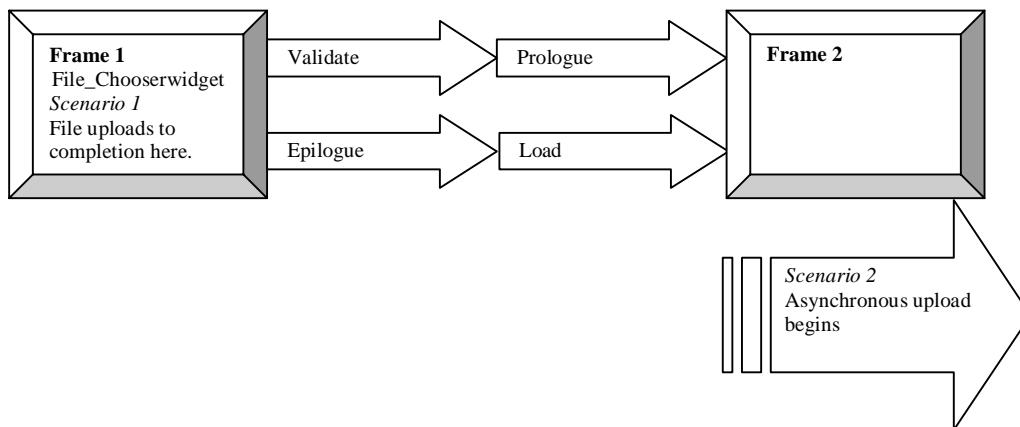
```
upload sourcefile /reports/report.doc targetfile report.doc delete;
```

Upload Command vs. Widget Upload Option

The upload file widget option and the upload command are intended to solve two different problems:

- The file chooser with upload flag allows the user to upload one file before the server-side wizard programs are run. The file would generally contain data to be parsed and used by the wizard. For example, you could populate an object's attributes or even a wizard list box based on a file selected by the wizard user.
- The upload command allows the user to upload one or more files for server side processing. For example, you could preserve the client directory structure during file checkin so that a checkout on a different client would maintain the same relative directories (such as for use with CAD files that generally contain an inherent directory hierarchy). This could be achieved by parsing the directory structure and list of files from a user selected text file (using a wizard file chooser with upload option), and then uploading the files for the server-side checkin.

When running a wizard with eMatrix Web Navigator, control passes between the applet running on the client (drawing frames, accepting user input) and the Collaboration Kernel running on the server (executing wizard programs and their embedded MQL commands). The frame transition programs (load, validate, epilogue, prologue) run on the server, and control is not returned to the client until they have all been run. Since the upload/download commands are client-side tasks that need to be controlled from the client, they are not processed synchronously within the execution of the wizard program. Rather, they are processed when control next returns to the client; that is, after all the frame transition programs have run to completion. Consider the following scenarios, illustrated below:



- *Scenario 1:* File_Chooser widget has the upload flag set so, before the frame transition programs are run, the file selected on Frame 1 is uploaded and available to these programs.
- *Scenario 2:* File_Chooser widget does NOT have the upload flag set. Instead, the Epilogue has the command "mql upload ...". In this case, after ALL the frame transition programs are run, an asynchronous task, running in the background, is launched to upload the file selected on Frame 1, so it is NOT guaranteed to be available for Frame 2.

Files uploaded to the server are automatically deleted when the session that put them there is finished, if they have not been deleted before that programmatically.

Running/Testing the Business Wizard

Any eMatrix user with appropriate access can run a wizard program from within eMatrix Navigator or from the system command line, including from a Windows desktop icon. A wizard can also be launched from within a program.

MQL trace can be used to test timing and debug wizard programs. For information, see [Testing Timing and Debugging Programs](#) in Chapter 19.

A wizard can be run from within eMatrix by any of the following methods:

- including its name in an eMatrix Toolset (see [Toolsets Defined](#) in Chapter 42)
- executing the wizard as a Method, which requires an object as its starting point
- executing the wizard from within a program, using the `appl` command

.Command Line Interface

From the system command line, the syntax of the command to run a wizard program is:

```
matrix -wizard NAME
```

where NAME is the name of the wizard program to launch.

The syntax of the command to run a wizard as a method is:

```
matrix -wizard businessobject OBJECTID WIZARD_NAME
```

OBJECTID is the OID or Type Name Revision of the business object.

where WIZARD_NAME is the name of the wizard program to launch.

When a wizard is run from the system command line, the user sees only the wizard user interface and is insulated from the rest of the eMatrix user interface. The eMatrix application is started, and when the wizard code has been executed, the eMatrix application shuts down. For this reason, a Business Administrator testing the wizard throughout its creation process would probably test from within eMatrix Navigator instead of using this method.

For example, a user may have to check in a report each week. A wizard could be created to gather information such as the name and type of the report and the user/department submitting it. Because the user does not need any of the other features of eMatrix to check in the report, the wizard could be run from the system command line or a desktop icon in Windows.

eMatrix Interface

A wizard can also be run from within eMatrix Navigator by either including its name in an eMatrix Toolset or by executing the wizard as a Method, which requires an object as its starting point.

When a program or wizard is added as a method to a type, then that program or wizard needs a business object in order to execute even if the Needs Business Object option was not enabled for that program/wizard.

To include a wizard name within a toolset definition, see [Creating Toolsets](#) in Chapter 42.

The Toolset button will appear in the eMatrix Navigator toolbar only when the Toolset is defined as active (the default).

Invoking a Wizard from a Program

A wizard can be invoked from within a program, using one of the following `appl` commands:

```
appl wizard [bus OBJECTID] WIZARD_NAME [{ARG}];  
appl wizard [set SETNAME] WIZARD_NAME [{ARG}];
```

where:

`OBJECTID` is the OID or Type Name Revision of the business object.

`SETNAME` is the name of a set.

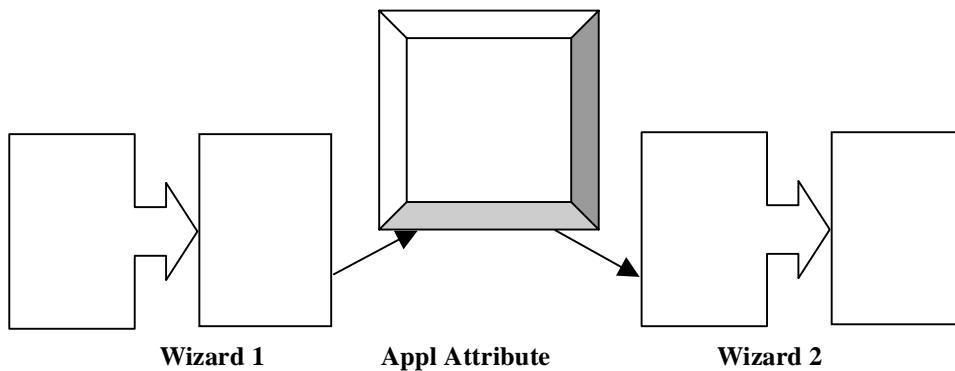
`WIZARD_NAME` is the name of the wizard that is being launched.

`ARG` is zero or more space-delimited strings that are passed to the program as arguments.

When a business object or set is specified, the `SELECTEDOBJECTS` macro is populated with the object IDs (OIDs). If arguments are included, they are passed in the `WIZARDARG` macro.

This, together with the ability to launch a program when an application dialog is dismissed (see [Application Command](#) in Chapter 19), allows the ‘linking’ of several small wizards or frames together with eMatrix ‘appl’ dialogs into a powerful aggregate. Wizard developers can present the components wizards/frames in any order desired, communicating between them via their passed arguments.

For example, if you wanted an attributes dialog to be launched in the middle of a wizard, you would actually create two smaller wizards and use the `appl attributes` command with a dismiss program as shown in the diagram below. The dismiss program is run whenever the dialog that it is applied to is dismissed. The dismiss program does not override the behavior of any buttons included in the dialog; rather it executes the program after database activity associated with the button is complete.



Wizard 1 has the following in its epilogue:

```
appl attributes bus Assembly XR7 A program LaunchWizard2;
```

After Wizard 1’s code is run, the attribute dialog is displayed for the object. When the attribute dialog is dismissed, the program `LaunchWizard2` is called. `LaunchWizard2` includes the following command:

```
appl wizard wizard2;
```

which starts the second wizard.

The RPE macros for `TYPE`, `NAME`, `REV`, `OBJECTID`, `OBJECT`, and `SELECTEDOBJECTS` are set to the appropriate values and passed to `LaunchWizard2` program (refer to the [Macros Appendix](#) for more information). Additional values can be

passed as program arguments (such as attribute values.) For Relationshipattributes and Relationshiphistory dialogs, only CONNECTIONID is populated.

Limitations

The `appl wizard` command can be invoked *only* from top level programs or methods (including the wizard code that is executed from the finish button), and *never* from within user-defined transaction boundaries. Invoking a wizard from another program or from within an active transaction will produce an error. The reasons for these limitations are:

- To prohibit implementations from encouraging user interaction while a transaction is open. This is important because when a transaction is open, there are likely to be rows or tables in the database that are locked by the transaction until it is completed (committed or aborted). During this time, other users' activities could be put in a wait mode awaiting the release of the locks, which could be an indefinite time.
- To prevent nesting of wizards. This would add significant complexity to the processing of wizards, such as in determining which wizard's frame is next. The capability of linking wizards end-to-end provides significant power and flexibility without unnecessary complexity.
- To provide well-defined RPE behavior. A wizard launched by `appl wizard` receives the same RPE as it would if it were launched directly from the toolbar or as a method. This is guaranteed since the launch of the invoked wizard has been explicitly delayed to occur after the calling program has exited and its local RPE has been cleared, so that there is NO sharing of local RPE values between the program that launches the wizard and the wizard itself. If information needs to be passed from the calling program to the wizard, it must be passed via the wizard arguments or the global RPE.

With these considerations in mind, `appl wizard` is allowed only if the following are true. All other cases will generate an error.

- There is no transaction open.
- INVOCATION must be one of the following: method, program, or action, as long as the action is also deferred so that it is outside the transaction. (The "body" code of a wizard has an INVOCATION value of "program" or "method" depending on how it was launched.)

It follows then, that INVOCATION cannot be any of the following: format, check, override, range, load, validate, prologue, epilogue, button, autoactivity, or action that is not deferred.

- The invoking program is not a nested program, meaning that the invoking program cannot be the result of an `exec prog` or an `exec bus T N R` method invocation from a higher-level program.

Note that `appl wizard` can be used in programs that are "uploaded" into a program that meets the above criteria, since a uploaded program is actually run as part of the program that uploads it.

Copying and/or Modifying a Business Wizard Definition

Copying (Cloning) a Wizard Definition

After a wizard is defined, you can clone the definition with the Copy Wizard statement. This statement lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy wizard SRC_NAME DST_NAME [MOD_ITEM] {MOD_ITEM} ;
```

SRC_NAME is the name of the wizard definition (source) to copied.

DST_NAME is the name of the new definition (destination).

MOD_ITEMS are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

MOD_ITEMS are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

Modifying a Wizard Definition

After a wizard is defined, you can change the definition with the Modify Wizard statement. When modifying a wizard program that is used to launch an application, however, consider upward and downward compatibility between software versions.

The following statement lets you add or remove defining clauses and change the value of clause arguments:

```
modify wizard NAME [MOD_ITEM] {MOD_ITEM} ;
```

NAME is the name of the wizard you want to modify.

MOD_ITEM is the type of modification you want to make.

You can make the following modifications. Each is specified in a Modify Wizard clause, as listed in the following table. Note that you only need to specify fields to be modified:

Modify Wizard Clause	Specifies that...
code CODE	The current code definition changes to that of the new code entered.
description VALUE	The current description, if any, changes to the value entered.
external mql	The specification of the wizard program type (external or MQL) changes as entered.
file FILENAME	The current file name changes to the new name entered.
icon FILENAME	The image is changed to the new image in the file specified.
name NAME	The current name of the program changes to the name entered.

Modify Wizard Clause	Specifies that...
[!]needsbusinessobject	<p>The status of the need for a business object changes as indicated here:</p> <p>needsbusinessobject is used when a business object is needed.</p> <p>!needsbusinessobject is used when a business object is not needed.</p>
[!]downloadable	<p>The status of downloadable changes as indicated here:</p> <p>downloadable is specified when the program includes code for operations not supported on the Web product (for example, Tk dialogs or reads/writes to a local file).</p> <p>!downloadable is specified when the program does not include code for operations not supported on the Web product.</p>
execute immediate	The status of wizard execution changes so the program runs within the current transaction.
execute deferred	The status of wizard execution changes so the program runs only after the outermost transaction is successfully committed.
hidden	The hidden option is changed to specify that the object is hidden.
nothidden	The hidden option is changed to specify that the object is not hidden.
frame FRAME_NAME FRAME_MOD_ITEM {FRAME_MODE_ITEM}	The named frame item(s) are changed to the new item(s) specified.
add FRAME FRAME_NAME [before FRAME] [FRAME_ITEM {FRAME_ITEM}]	The named frame is added.
remove FRAME FRAME_NAME	The named frame is removed.
property NAME [to ADMINTYPE NAME] [value STRING]	The named property is modified.
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.

As you can see, each modification clause is related to the arguments that define the wizard frame.

Modifying Wizard Frames

The following subclauses are available to modify the existing frames in a wizard:

Modify Wizard Frame Subclauses	Specifies that...
FRAME_ITEM	The named frame item(s) are changed to the new item(s) specified.
name FRAME_NAME	The frame name is changed to the new name specified.
widget WIDGET_NAME WIDGET_ITEM {WIDGET_ITEM}	The existing widget is modified with the WIDGET_ITEM(S) specified.
add widget WIDGET_TYPE [WIDGET_ITEM {WIDGET_ITEM}]	The named widget is added.
remove widget WIDGET_NAME	The named widget is removed.

Modifying Widgets

The following subclauses are available to modify the existing widgets in a wizard frame:

Modify Widget Subclauses	Specifies that...
name WIDGET_NAME	The widget name is changed to the new name specified.
value VALUE [selected value input ARG_STRING]	The existing widget value is modified.
load PROG_NAME [input ARG_STRING]	The load program for the named widget is changed.
validate PROG_NAME [input ARG_STRING]	The validate program for the named widget is changed.
color FOREGROUND [on BACKGROUND]	The foreground and/or background color(s) of the named widget are changed to the new color(s) specified.
start XSTART YSTART	The position of the named widget is changed.
size WIDTH HEIGHT	The dimensions of the named widget are changed.
font FONT_NAME	The font for the text in the named widget is changed.
autoheight [false true]	The value of autoheight for the named widget is changed. True indicates that the height of the widget is determined by the system; false indicates that the height of the widget is coded.
autowidth [false true]	The value of autowidth for the named widget is changed. True indicates that the width of the widget is determined by the system; false indicates that the width of the widget is coded.
drawborder [false true]	The drawborder value for the named widget is changed. True indicates that the widget will have a border; false indicates that the widget will not have a border.

Modify Widget Subclauses	Specifies that...
multiline [false true]	The multiline value for the named widget is changed. True indicates that a text box widget can contain multiple lines or a list box widget can allow multiple selections; false indicates that the widget will contain a single line or allow a single selection.
edit [false true]	The edit value for the named widget is changed. True indicates that the widget field is editable by the user; false indicates that the field is not editable.
scroll [false true]	The value of the scroll option for the named widget is changed. True indicates that the widget field is scrollable; false indicates that the field is not scrollable.
password [false true]	The password value for the named widget is changed. True indicates that a password is required; false indicates that no password is required.
upload [false true]	The upload value for the named file text box widget is changed. True indicates that the widget is designed to allow the user to select a client-side file and copy it to the server; false indicates that the wizard programmer will include the MQL upload command in one of the wizard programs.

As you can see, each modification clause is related to the arguments that define the wizard frame widgets.

Deleting a Business Wizard

If a wizard is no longer required, you can delete it with the Delete Wizard statement:

```
delete wizard NAME;
```

NAME is the name of the wizard to be deleted.

When this statement is processed, eMatrix searches the list of wizards. If the name is not found, an error message is displayed. If the name is found, the wizard is deleted.

Working With Programs

Overview of Programs.....	457
Creating a Program	458
Code Clause	458
Description Clause.....	461
Java or MQL or External Clause.....	461
File Clause.....	462
Rule Clause	462
Icon Clause.....	462
Needs Business Object Clause	462
Downloadable Clause.....	463
Execute Clause.....	463
Hidden Clause	464
Piped Clause	464
Pooled Clause	465
Property Clause	465
Handling Variables.....	467
Using the Runtime Program Environment	467
Macro Processing and Command Syntax	470
Programs for Attribute Ranges	472
Transaction Management.....	473
Transaction Boundaries.....	473
Java Program Objects.....	474
Java Command.....	474
Compile command.....	475
Testing Timing and Debugging Programs	476
Program Strategies.....	477
Nesting Programs	477
Notifying Users (Notice, Warning and Error Commands).....	477
Writing Piped Program Code	478
Program Object Select Expressions	481
Program With Arguments	482
Programming for Use in a Loosely-Coupled Database	484
Programs Exported to XML	485
Programs Using Output Length	485
Optimizing Programs	486
Access Business Objects and Relationships by Vault or OID	486
Selecting Relationship and Business Object Data.....	487
Referencing Schema Object Names	488

Query Performance	488
Avoid Initializing multiple Tcl Interpreters	488
Using Transaction Boundaries	489
Storing “Static” Data	489
Program Exit vs. Program Failure	490
Generating an Exit Code	491
Program Failure	492
Dialog Scripting Language.....	493
Application Command	493
Application Help	497
Using the Application Command	497
Programming for the Web Version	500
Downloadable Programs	500
Downloadable Program Threads Share Context	500
Tcl Programs	503
File Checkin and Checkout	503
Dialog Scripting on the Web	504
Web URL Access	504
Executing a Program	505
Command Line Execution	505
Dealing With User Context	506
Push Context Command	507
Pop Context Command	507
Initialization File Context Variable	507
Program Output.....	508
Copying and/or Modifying a Program Definition	510
Copying (Cloning) a Program Definition	510
Modifying a Program Definition	510
Deleting a Program	513

Overview of Programs

A *program* is an object created by a Business Administrator to execute specific commands. Programs are used:

- In format definitions for the edit, view, and print commands.
- As Action, Check, or Override Event Triggers, or as actions or checks in the lifecycle of a policy.
- To run as *methods* associated with certain object types. (Refer to [Working With Types](#) in Chapter 13, for the procedure to associate a program with a type.)
- In Business Wizards, both as components of the wizard and to provide the functionality of the wizard.

A program might execute operating system commands. This type of program is *external*. Examples are programs such as a word processor or a CAD design program which can be specified as the program to be used for the edit, view, and print commands in a format definition.

Other programs might use only MQL/Tcl commands. For example, a check on a state might verify the existence of an object using an MQL program.

Depending on how commands are executed (either from within a program object or typed directly into the MQL console) and what mode is active (MQL or Tcl) slightly different syntax may be required. Refer to [Macro Processing and Command Syntax](#) for more information.

Some programs may require a business object as the *context* or starting point of the commands. An example of this is a program that connects a business object to another object.

Creating a Program

A program is created with the Add Program statement:

```
add program NAME [ADD_ITEM {ADD_ITEM}];
```

NAME is the name you assign to the program. The program name is limited to 127 characters. For additional information, refer to [Business Object Name](#) in Chapter 34.

ADD_ITEM is an Add Program clause which provides additional information about the program. The Add Program clauses are:

code CODE
description VALUE
external mql java
file FILENAME
icon FILENAME
rule NAME
[! not]needsbusinessobject
[! not]downloadable
execute immediate deferred
[! not]pipe
[! not]pooled
[! not]hidden
property NAME [to ADMINTYPE NAME] [value STRING]

All of these clauses are optional. You can define a program by simply assigning a name to it. You will learn more about each Add Program clause in the sections that follow.

Code Clause

The Code clause of the Add Program statement defines the commands for the program. Below are examples of program code that could be written to provide various functionality. See [Handling Variables](#) for additional information.

Legal characters in XML are the tab, carriage return, line feed, and the legal graphic characters of Unicode, that is, #x9, #xA, #xD, and #x20 and above (HEX). Therefore, other characters, such as those created with the ESC key, should not be used for ANY field in eMatrix, including business and administrative object names, description fields, program object code, or page object content.

Format Definition Example Program

To be used in a format definition, a program object definition must include these characteristics:

- The needsbusinessobject clause must be true.
- The code clause must contain the command needed to execute the program and the syntax for the command must be appropriate for the operating system.
- The code clause should end with the \${FILENAME} macro so the program opens any file. Enclose the macro in quotes to ensure that files with spaces in their names are opened correctly.

To launch Microsoft Word 97 or 2000 on Windows and open a document file for viewing and editing, the following program code might be used in the external program named MSWORD:

```
${PROG} /w "${FILENAME}"
```

The \${PROG} macro returns the command needed to execute the program defined by the file association mechanism of windows. The /w is needed for Word 2000, but is ignored for other versions.

For a simple generic format that uses file associations, do not define any programs for the edit, view and print clauses of the format. For a more complex and flexible generic Windows format that will open any file for view, edit, or print based on its file association, include a condition exception for Word. For example:

```
tcl;
eval {
... parsing code to take out quotes / args / etc. from the
registry
if [string match *winword* ${PROG}] {
exec [${PROG} /w "${FILENAME}"]
} else {
exec [${PROG} "${FILENAME}"]
}
}
```

On a UNIX system, you might use the following for a text format:

```
edit textedit "${FILENAME}"
```

Note that the quotes allow the file name to contain spaces.

Action Program Example

Program Name: Action to connect to Assembly 444 1
Description: Used in the Auto Policy
MQL Program
Needs Business Object

You might define a program to be used as an action on a State as follows:

```
tcl
if (${EVENT} == "promote") {
mql connect bus ${OBJECTID}
relationship [Parts] from Assembly 444 1;
}
```

This code would only be executed when the most recent action (upon arrival into the state) is “promote.” If the object reached the state a second time (by demotion), the object would not be connected a second time. EVENT can have the following values at this point: promote, demote, create, or revisioned. Additional code could be provided

for each case so that actions appropriate for each possibility could be automated by the one program. For example, to delete all signatures on a demote event, the following could be added to the above program:

```
if (${EVENT} == "demote") {  
    mql unsign bus ${OBJECTID} signature all;  
}
```

Note that it is recommended that Actions and Checks are configured as promote Triggers, and not as Lifecycle Checks and Actions. Refer to [Working With Event Triggers](#) in Chapter 20 for more information.

Check Program Example

Program Name: Drawing Object Exists?

Description: Used as a check in policy for solid model objects

MQL Program

Needs Business Object

Programs for checks must return either a true or a false value. An example is to check for the existence of a business object of the same name as the current one but with a different type:

```
print bus Drawing ${NAME} ${REVISION}  
exists;
```

If the drawing exists, the check is satisfied. If it does not exist, the check is not satisfied.

Note that it is recommended that Actions and Checks are configured as promote Triggers, and not as Lifecycle Checks and Actions. Refer to [Working With Event Triggers](#) in Chapter 20 for more information.

Creating a Program for Execution as Needed

Program Name: Engineering Assemblies

MQL Program

Does Not Need Business Object

You can create programs that users may execute as needed. For example, you might define a program that creates a table for the Engineering Department. You would then associate the program (method) with a type to which the Engineering Department has

access, such as Assemblies. Since tables are personal settings (each user must define their own tables), the program could be executed by each engineer. Sample program code is:

```
add Table Budget Differential
    units inches
    description "Cost Analysis"
    column label "Estimated Cost"
        businessobject attribute ["Estimated Cost"]
        autoheight
        autowidth
    column label "Actual Cost"
        businessobject attribute ["Actual Cost"]
        autoheight
        autowidth
    column label "Budget Differential"
    businessobject attribute ["Estimated Cost"] ?
        attribute["Actual Cost"]
        autoheight
        autowidth;
```

Description Clause

The Description clause of the Add Program statement provides general information for you and the user about the commands associated with this program and the overall function of the program. There may be subtle differences between programs; the description can point out the differences.

There is no limit to the number of characters you can include in the description. However, keep in mind that the description is displayed when the mouse pointer stops over the program in the Program chooser. Although you are not required to provide a description, this information is helpful when a choice is requested.

For example, assume you want to write descriptions for two word processing programs. For the program named “BestWord” you might assign a description such as:

```
add program BestWord
    description "Use to launch the BestWord word processing application"
```

Descriptions help to clarify the purpose of the program. You may enter a string of any length.

Java or MQL or External Clause

You can specify the type of program as `java` or `mql` or `external`.

A Java Program Object (JPO) is just another type of Program object that contains code written in the Java language. Generally, anywhere a Program object can be used, a JPO can be used. See [Java Program Objects](#) for details.

An MQL program can be run from any machine with eMatrix installed; it is not necessary for MQL to be available on the machine. If not specified, `mql` is the default.

```
add program NAME mql;
```

An external program consists of commands that are evaluated by the command line syntax of the machines from which they will be run. When creating external programs, remember that the commands that you enter will be evaluated at each eMatrix user’s workstation as if they were being typed at the operating system’s command prompt. Be sure that the users have the appropriate application files available from their workstation. External program

objects may also be defined as “piped,” providing a built-in MQL command line service to handle standard input and output. Refer to [Piped Clause](#) for more information.

Since MQL can be launched from a command line, MQL code could be specified in an external program. This would spawn a separate MQL session that would run in the background. In this case, MQL would have to be installed on every machine that will run the program.

```
add program NAME external;
```

A Java program has code written in the Java language. A Java program can be run anywhere a program object can be used.

```
add program NAME java;
```

File Clause

The File clause enables you to specify a file that contains the code to be used in the program.

```
add program NAME file FILENAME;
```

Rule Clause

Rules are administrative objects that define specific privileges for various eMatrix users. The Rule clause enables you to specify an access rule to be used for the program.

```
add program NAME rule RULENAME;
```

Icon Clause

The Icon clause associates a special image with a program. When a user searches for a program, the icon can help identify the object to select. Icons help users locate and recognize items. You can assign a special icon to the new program or use the default icon. The default icon is used when in view-by-icon mode. Any special icon you assign is used when in view-by-image mode. When assigning a unique icon, you must use a GIF image file. Refer to [Icon Clause](#) in Chapter 1 for a complete description of the Icon clause.

GIF filenames should not include the @ sign, as that is used internally by eMatrix.

Needs Business Object Clause

You can specify that the program must function with a business object. For example, you would select this option if the program promotes a business object. If, however, the program creates a business object, the program is independent of an existing object and this option would not apply.

```
add program NAME needsbusinessobject;
```

eMatrix runs any program specified as “needs business object” with the selected object as the starting point. If a method does not use a business object, the selected object would not be affected.

If not set, some macros, including [Business Object Identification Macros](#), will not be available.

The `doesneedcontext` selectable is available on programs to determine this setting in an existing program.

The following indicates that a business object is not needed:

```
add program NAME !needsbusinessobject;
```

When defining a type or format, you can specify program information:

- When defining a type, you can indicate any defined programs. Even programs that do not require a business object could be associated with a type in order to make them available to users.
- When defining a format, only the programs defined as “needs business object” would be appropriate for the view, edit, and print procedures since eMatrix will pass a file from a business object to the program.

Downloadable Clause

If the program includes code for operations that are not supported on the Web product (for example, Tk dialogs or reads/writes to a local file) you can include the `downloadable` clause. If this is included, this program is downloaded to the Web client for execution (as opposed to running on the Collaboration Server). For programs not run on the Web product, this flag has no meaning.

```
add program NAME downloadable;
```

If the `downloadable` clause is not used, `notdownloadable` is assumed.

Due to the restriction that downloaded programs must execute in a deferred mode, there are several cases that need to be addressed by system logic. If just the `downloadable` clause is given, then `deferred` is assumed (see [Execute Clause](#)). If the `downloadable` clause is given, and the `execute` clause is `immediate`, an error will be generated. Likewise, if on program modification command a mismatch occurs, an error will be generated that reads:

A program that is downloaded cannot execute immediately.

Note that the `usesexternalinterface` clause continues to be supported for historical reasons. Scripts and programs that use this clause result in Program objects which have their `execute` flag set to `deferred` and their `downloadable` flag set.

Java Program Objects cannot be downloadable.

Execute Clause

Use the `execute` clause to specify when the program should be executed. If the `execute` clause is not used, `immediate` is assumed.

`Immediate` execution means that the program runs within the current transaction, and therefore can influence the success or failure of the transaction, and that all the program's database updates are subject to the outcome of the transaction.

`Deferred` execution means that the program is cued up to begin execution only after the outer-most transaction is successfully committed. A deferred program will not execute at all if the outer transaction is aborted. A deferred program failure affects only the new isolated transaction in which it is run (the original transaction from which the program was launched will have already been successfully committed).

For example, to defer the execution of the “Roll up Cost” program:

```
modify program "Roll up Cost" execute deferred;
```

However, there are a number of cases where deferring execution of a program does not make sense (like when it is used as a trigger check, for example). In these cases the system will execute the program immediately, rather than deferring it until the transaction is committed.

There are four cases where a program’s execution can be deferred:

- Stand-alone program
- Method
- Trigger action
- State action

There are six cases where deferred execution will be ignored:

- Trigger check
- Trigger override
- State check
- Range program
- Wizard frame prologue/epilogue
- Wizard widget load/validate

There is one case where a program’s execution is always deferred:

- Format edit/view/print

A program downloaded to the Web client for local execution (see [Downloadable Clause](#)) can be run only in a deferred mode. Therefore, if you use the downloadable option, program execution is automatically deferred.

Note that the usesexternalinterface clause continues to be supported for historical reasons. Scripts and programs that use this clause result in Program objects which have their execute flag set to deferred and their downloadable flag set.

Hidden Clause

You can specify that the new program is “hidden” so it does not appear in the Program chooser in eMatrix, which simplifies the end-user interface. Many programs are not intended to be executed as stand-alone programs (such as nested programs) and users should not be able to view these program names in the eMatrix Program chooser. Users who are aware of the hidden program’s existence can enter its name manually where appropriate. Hidden objects are also accessible through MQL, but printing a hidden program is not possible unless you are an eMatrix Business or System Administrator.

Piped Clause

You can specify that external program objects use the “piped” service. Piped programs may use a built-in MQL command line service to handle standard input and output. When piped is specified, External is assumed. The piped service is not available to MQL program objects. Execution may be immediate or deferred. Piped programs cannot be downloadable. Refer to [Writing Piped Program Code](#) for more information.

Java Program Objects cannot be piped.

Pooled Clause

Each time an MQL program object runs Tcl code and then exits out of Tcl mode, a Tcl interpreter is initialized, allocated, and then closed. During an eMatrix session, you may execute several programs, and one program may call other programs, all of which require a Tcl interpreter and therefore the overhead of its use. In an effort to optimize multiple Tcl program execution, Tcl program objects may be specified as “pooled.” When such a program is first executed in a session, a pool of Tcl interpreters is initialized, one of which is allocated for the executing code. When the code is completed, the interpreter is freed up. Subsequent Tcl code that is executed in a pooled program during the session will use an interpreter from the already initialized pool.

When programs are created, the default is that they are not pooled. To define or modify an MQL type program to use the pool of interpreters, use the following syntax:

```
mql<> add|modify program PROG_NAME [!]pooled;
```

The “!” may be used to turn off the `pooled` setting of a program.

The number of interpreters available in a session is controlled by the `MX_PROGRAM_POOL_SIZE` setting in the initialization file (`matrix.ini`, `bos.ini`, or `ematrix.ini`). `MX_PROGRAM_POOL_SIZE` sets the initial size of the Tcl interpreter pool. This setting is also used to extend the pool size when all the interpreters in the pool are allocated and another is requested. The default is 10.

Usage

Enabling the Tcl interpreter benefits MQL/Tcl programs that are nested or run in a loop, such as the trigger manager. Also, while wizards do not support the pooled setting, the Tcl programs that make up a wizard (load, validate, etc.) should make use of an interpreter pool to optimize performance.

Unexpected results may occur if the pooled setting is turned on in a program without first reviewing and validating its code. Good programming techniques must be adhered to ensure proper results. When using an interpreter pool, Tcl variables are not cleared before freeing up an interpreter. This means that programs must explicitly set variables before using them, in case a previously executed program made use of the same variable name.

External, downloadable and Java programs do not use the Tcl interpreter pool, regardless of the setting in the program definition. In addition, MQL/Tcl programs that use the TK toolkit will not benefit from using the pooled setting, since user interaction is required. In fact, unexpected results, such as leaving a TK dialog displayed, may occur due to the use of variables as described above.

Before modifying existing programs to use the pooled setting, the code should be reviewed and validated. Only programs that include tcl code but no TK code should use the pooled setting.

Property Clause

Integrators can assign ad hoc attributes, called Properties, to the program. Properties allow associations to exist between administrative definitions that aren't already associated. The property information can include a name, an arbitrary string value, and a reference to another administration object. The property name is always required. The value string and object reference are both optional. The property name can be reused for different object

references, that is, the name joined with the object reference must be unique for any object that has properties.

```
add program NAME  
    property NAME [to ADMINTYPE NAME] [value STRING];
```

For additional information on properties, see *Overview of Administration Properties* in Chapter 24.

Handling Variables

Because the operating system is unaware of the business objects stored in eMatrix, a method is needed to pass information from eMatrix to the command line in order to make the commands useful. In early versions of eMatrix, an internal structure called a macro was used to pass values as variables between eMatrix and the operating system. In eMatrix version 6, the Runtime Program Environment (RPE) was added to extend the support for dynamic information.

Macros continue to be supported. Refer to the [Macros Appendix](#) for more information.

Using the Runtime Program Environment

The *Runtime Program Environment* (RPE) is a dynamic heap in memory used to hold name/value pairs of program environment variables. The RPE makes it possible to pass parameters between internal programs and scripts. External programs cannot access the RPE.

The RPE is initialized when the eMatrix application starts and terminates when the eMatrix application ends. Variables placed in RPE memory are generally accessible only for the duration of the outermost program. Variables can be made available for the duration of the application if the `global` keyword is used. A variable can be removed from the RPE at any time.

RPE and MQL

The RPE can hold, in addition to input arguments, any arbitrary variables that a user chooses to define. The MQL “environment” set of commands is used for defining and managing RPE variables. These commands are accessible from the Tcl portion of MQL, and allow a program to write, read, delete, and list program environment variables.

Note that even though the program environment is accessible in Tcl mode (since this is where variable manipulation normally takes place), the program environment is managed by eMatrix. Note also that program environment variables are lost when the session ends. The commands are:

Runtime Program Environment Commands		
Mql Command	Arguments	Comment
<code>set env</code>	<code>[global] VARNAME VALUE</code>	Write variable to the RPE.
<code>get env</code>	<code>[global] VARNAME</code>	Read variable from the RPE.
<code>unset env</code>	<code>[global] VARNAME</code>	Delete variable from the RPE.

Runtime Program Environment Commands		
list env	[global]	List variables in the RPE.
clear env		Delete all variables from the RPE.
exists env	[global] VARNAME	Checks if an RPE variable exists. If the variable exists, "1" is returned, otherwise "0" is returned.
listnames env	[global]	Returns just the names (not values) of all global RPE variables.

When macros are filled during the launch of a Program object, the system places the existing set of macros into the RPE, so that they can be used by running programs.

Notes:

- The `set env` command will overwrite NAME with a new value if it already exists.
- The `get env` command returns VALUE (i.e., prints to stdout).
- The `list env` command prints all of the NAME=VALUE pairs.
- All RPE command outputs are sensitive to the quote `on|off` command.
- Only a System Administrator can issue the `clear env` command.
- The keyword `global` (explained later) is optional.
- VARNAME and VALUE are case sensitive, and need to be quoted if they contain spaces.

RPE and Tcl are closely integrated: you can set Tcl variables with RPE variables (using `set var [mql get env NAME]`) and set RPE variables with Tcl variables (using `mql set env NAME $var`). Of course, the VARNAME field itself can also be a Tcl variable.

In order to reduce the potential build-up of variables over time (if they are not explicitly removed from the RPE), the RPE has been divided into two different name spaces:

- **local** — cleared automatically by the system when the system reaches the idle state.
- **global** — never cleared by the system.

The **local** name space will keep variables around temporarily, but long enough for all nested programs/scripts to complete.

The **global** name space will keep variables around until specifically removed by the `unset` command. Of course, both name spaces go away when the application terminates.

When using MQL, it makes no sense to set a local RPE variable at the MQL command prompt (what is sometimes called the “idle state”). This is due to the fact that the local name space is cleared automatically by the system when the next MQL command prompt appears. Therefore, the system will automatically place RPE variables set at the MQL command prompt into the global name space (even if the `global` keyword is not given).

For example:

```
MQL<6> set env SITE Chicago;
```

```
MQL<7> set env global COURSE "Intro to MQL";  
MQL<8> list env;
```

will result in both SITE and COURSE being in the list because both were placed in the global name space, which is not automatically cleared by the system. Note that it is good style (but certainly not required) to use all uppercase letters for global RPE variable names. Note also that the last command results in both name spaces being listed, but since the local name space is always empty at the command prompt, this command could have been `list env global` and resulted in the exact same list.

To summarize:

- When running MQL, the system is at the idle state when the MQL prompt appears. When running the GUI applications, the system is at the idle state when not specifically processing a request. (The message bars say “Ready.”)
- By default, the RPE commands operate on the local name space. The one exception is that any variables set at the idle state automatically go into the global name space, even if the `global` keyword is not supplied.
- When the `global` keyword is used, only the global name space is searched/updated.
- When fetching variables from the RPE, the local name space is searched first and then the global name space. This is how most programming languages behave. If a local and global variable share the same name, then one must use the `global` keyword to fetch the global variable.
- All local variables placed into the RPE (no matter the depth of nesting of the programs/scripts) stay in the RPE until explicitly removed or the idle state is reached.

Program Environment Variables

In using the RPE to pass data between program objects, careful attention must be paid to the naming of program environment variables and to the cleaning up of these variables. All data is passed in the form of strings. Here are some guidelines:

- The program name is automatically associated with a variable named “0”.

When calling a JPO method that returns a String, a RPE variable named after the JPO name is created and set to the value of the returned string. This is not the case when calling a JPO method that returns an integer. No such RPE variable is created.

- Program input parameters are named “1”, “2”, etc.
- The calling program can manually place the expected local variables into the RPE (using the naming scheme above) before executing the called program, or simply add them to the end of the execute command; the system will automatically place them in the RPE with the proper names.
- Programs should return their data in a variable with the name identified with the 1st input parameter.
- Lists should be returned using a Tcl form (space separated, with curly braces used to surround items that contain spaces or special characters).
- Since the RPE is shared memory, any variable can be overwritten by a variable of the same name. It is good practice to always capture program variables immediately into custom local variables to be assured that the values will be available where necessary within the program. RPE variables can be saved as local variables (in Tcl: `set`

`localUser [mql get env USER])` or saved back into the RPE under a different name that won't be overwritten (for example: `programName.USER` where `programName` is the name of the program).

Any eMatrix program or trigger that runs at any time before a program completes has the potential to alter RPE variables. For example, a user starts a wizard, then checks the attributes of another object in the database. Depending on how the object is configured, a program or trigger may fire automatically without the user's knowledge that could overwrite existing RPE variables.

A properly written program should *always* fetch out of the RPE all variables needed before proceeding.

For examples of using the Runtime Program Environment, see [Working With Business Wizards](#) in Chapter 18.

Macro Processing and Command Syntax

Macro processing involves the replacement of variable and macro names with their current values to create the precise code that is executed from a Program object or with the `shell` command. Macro values are not evaluated unless and until they are actually used. For example, a modify attribute trigger will not cause an attribute range program to be executed unless the macro `${ATTRCHOICES}` is explicitly used by the program.

Macros can affect the surrounding text, if the required characters have additional uses in programming, such as “\$” or “\”. Because of this, in order to be able to include a literal “\${” in your text, you must turn off the macro processing substitution on a case-by-case basis by using the escape character “\”.

For example, if `USER=guest` then:

```
output 'The value of \${USER} is ${USER}'
```

results in the output:

```
The value of ${USER} is guest
```

But one may also want the “\$” in the text to be macro processed. This can be accomplished by using a pair of backslashes. For example, if `MATRIXHOME=tools\matrix-9.0\` then:

```
output 'The file can be found in \\${MATRIXHOME}'
```

results in the output:

```
The file can be found in \tools\matrix-9.0\
```

So, to restate, macro processing of a backslash ():

- followed by a “\$”, results in a literal \$
- followed by another backslash, results in a single backslash
- in all other cases is treated literally

The consequence of this means that any backslashes that need to appear in pairs (like the beginning of a UNC path) will require escaping twice by using four backslashes. For example, one might use:

```
shell '\\\\ourserver\Matrix9\bin\winnt\mql.exe -v';
```

Note that macro processing is done in a single pass. Therefore, the values substituted in for macro names are not subject to macro processing. For example, if:

`MATRIXHOME=\\matrix9`, then one could use:

```
shell '$MATRIXHOME}\bin\winnt\mql.exe -v';
```

Also note that Tcl also recognizes the backslash (among other characters) as an escape character. This means any Tcl code in a Program may require additional escaping. Refer to Tcl documentation for information on escaping in Tcl.

MQL and Tcl

Because of macro processing (as discussed above), parsing of Tcl statements containing backslashes within an MQL program object is different than when those same statements are run from an MQL command prompt.

For example, create a text file with the command lines:

```
tcl;
set a "one\\two\\three"
puts "a contains $a"
puts "b will use slashes instead of backslashes"
regsub -all {\\"} $a {/} b
puts "now b contains $b"
```

Run this file from MQL (or alternatively type each line in MQL command window). The final line of output will be:

```
now b contains one/two/three
```

This is consistent with how the same lines of code behave in native Tcl.

However if you use this same code as an MQL program object, to get the same output you need to use triple rather than double backslashes so your code becomes:

```
tcl;
set a "one\\\\two\\\\\\three"
puts "a contains $a"
puts "b will use slashes instead of backslashes"
regsub -all {\\\\\} $a {/} b
puts "now b contains $b"
```

Programs for Attribute Ranges

As described in [Ranges Defined Within a Program](#) in Chapter 12, ranges can be assigned to attributes that explicitly define one or more legal values. A Program may be assigned as a range for a string attribute. This program object is used to produce a different set of legal string values based on the state or settings of the specific object or relationship in which the attribute is used. It is important to keep in mind that these programs are executed under two sets of circumstances:

- When the attribute range values are provided for selection (when the list box presents valid value choices);
- When the validity of a values is checked (when the **Modify** button is pressed in the attributes dialog, or the MQL `check attribute` command executed).

Code must be included to handle both circumstances.

The macros available to Range programs are listed in the [Macros Appendix](#).

Example

The following is an example of a program that can be used to define ranges:

```
add program nameRange
mql
  code 'tcl;
eval {
  set event [mql get env EVENT]
  set names {Larry Curly Moe}
  if { $event == "attribute choices" } {
    set output [mql get env 0]
    mql set env global $output $names
  } else {
    set value [mql get env ATTRVALUE]
    if {[lsearch -exact $names $value] == -1} {
      exit 1
    } else {
      exit 0
    }
  }
}'
```

Transaction Management

A *transaction* involves accessing the database or producing a change in the database. A transaction is defined as a *unit of work* which needs to succeed or fail as a whole. This, along with locking, ensures database consistency. There are two types of transaction locks:

- **Readonly locks:** Locks used to reference data.
- **Update locks:** Locks used to add, delete, or modify data.

While the eMatrix transaction model provides concurrency, it also puts much of the responsibility for transaction management on MQL/Tcl programmers. There are several things a programmer must keep in mind:

- *Minimize the use of explicit transactions.* While they guarantee consistent data, transactions place a demand on database resources and their corresponding locks impact concurrency.
- *Minimize the length of explicit transactions.* Locks are not released until a transaction is committed, so database resources may not be available to concurrent users.
- *Make transactions readonly whenever possible.* Read only transactions are less of a burden than update transactions, and with the MROW (Many Readers, One Writer) model this is especially the case. Update transactions, in addition to placing greater demands on the database, also can lead to accidental changes to the database.
- Many reads of the same data should be in a single transaction to take advantage of caching. To reduce I/O, data is cached whenever possible.
- Always match a `start transaction` with either a `commit` or `abort`.

See [Transaction Boundaries and Events](#) in Chapter 20 for more information. Also refer to *Working With Transactions* in Chapter 2.

Internally, eMatrix ensures database consistency by wrapping read transactions around *get* events, and update transactions around *set* events. These transactions are kept as short as possible to meet the goals given above. In some cases a number of these short transactions may be nested within a longer *outer* transaction. Nesting read transactions within an update transaction is not a problem, but the reverse is a problem to be avoided.

UPDATE transactions should NEVER be nested within a READ transaction.

Transaction Boundaries

MQL scripts are always run outside transaction boundaries, so proper transaction management should be carefully designed into the writing of any MQL/Tcl scripts.

As for Program objects, the **Execute** setting is checked to determine if the execution should be deferred until all open transactions are either committed or aborted.

Java Program Objects

A Java Program Object (JPO) is just another type of Program object that contains code written in the Java language. Generally, anywhere a Program object can be used, a JPO can be used.

JPOs have been created to satisfy a need for the ability to run Java programs natively inside the kernel, without creating a separate process and with a true object-oriented integration ‘feel’ as opposed to working in MQL. JPOs allow developers write Java programs on top of the eMatrix ADK programming interface and have those programs invoked dynamically.

When running inside the eMatrix Collaboration Kernel, the programs share the same context and transaction as the thread from which they were launched. In fact, the Java programs are running inside the same Java Virtual Machine (JVM) as the kernel.

For the current release, JPOs are not supported on the CORBA server.

Java Program Objects are also tightly integrated with the existing scripting facilities of eMatrix. Developers can seamlessly combine MQL, Tcl, and Java to implement their business model.

While Tcl will continue to be supported and does offer a scripting approach which can makes sense in some cases, the Java language brings several advantages over Tcl:

- compiled, and therefore greater run-time performance
- object-oriented
- thread safe

Java code must be contained in a class. In general, a single class will make up the code of a JPO. However, simply translating Tcl program objects into Java is not the goal. This would lead to many small classes, each containing a single method. The very object-oriented nature of Java lends itself to encapsulating multiple methods into a single class. The goal is to encapsulate common code into a single class.

A JPO can be written to provide any logical set of utilities. For example, there might be a JPO that provides access to Administration objects that are not available in the ADK. But for the most part, a JPO will be associated with a particular Business Type. The name of the JPO should match the name of the Business Type (but this is certainly not required).

It is the responsibility of the JPO programmer to manually create the JPO and write all of the methods inside the JPO. Keep in mind that JPO code should be considered server-side Java; no Java GUI components should ever be constructed in a JPO.

For details on writing JPOs, see the *Java Program Object Programmer’s Guide*.

Java Command

The `java` command has been added for JPOs. The syntax for this command is:

```
java CLASS_NAME [-method METHOD_NAME] [ARGS] [-construct ARG];
```

where ARGS is zero or more space-delimited strings and ARG is a single string. If the method returns an int, then this is assumed to be the exit code from the method (which has meaning to the eMatrix system when used as a trigger program, wizard utility program, etc.). If the method returns an Object, then the exit code is set to zero and the object is

serialized and passed back in the program's output stream. Obviously, this serialized object is of no use in the MQL environment, but is very useful in the ADK environment (see next section).

The `java` command takes any class name and finds that class both inside a JPO and outside the eMatrix database. The class name must be in its fully mangled form (see the *Java Program Object Programmer's Guide* for details about name mangling).

The `-construct` clause is used to pass arguments to the constructor (Note: if more than one argument needs to be passed to the constructor, the `-constructor` clause can be repeated as many times as necessary on a single command). This may be necessary to establish an object's identity. For example, if one had a JPO named "Project" that held business logic for managing business objects of type "Project," then this JPO would undoubtedly derived from the ADK BusinessObject class.

The `classname` selectable can be used on JPO programs to print the classname specified.

Compile command

You can compile a JPO after defining or modifying its code with the following MQL command:

```
compile prog PROGNAME_PATTERN [force];
```

Where `PROGNAME_PATTERN` can include wildcards.

`force` is used to recompile modified JPOs.

Explicitly compiling all JPOs ensures that the compilation succeeds and avoids the introduction of compilation delays for real-time users.

Value Chain Portfolio JPOs must be compiled before using the application.

For example, to compile all uncompiled JPOs, enter the following MQL command:

```
compile prog *;
```

To compile or recompile all JPOs, use:

```
compile prog * force;
```

Only Java programs can be compiled in this manner.

Testing Timing and Debugging Programs

Use the MX_MQL_TRACE ini setting to test timing and debug program objects and wizards. MX_MQL_TRACE can be set to true in any of the initialization files (matrix.ini, bos.ini, or ematrix.ini). The generated output includes information on any triggers that are fired and elapsed execution time. You can direct the output to stdout (via the MQL command line) or have the output display in an MQL window (desktop client). For the CORBA server, the output is displayed in the orbix log (UNIX server) or server window (NT server). In EJB, WebLogic echoes stdout to the startup window. Currently, there is no means of viewing stdout (or stderr) on the RMI Collaboration Server, since RMI activation in Java 2 runs without a console.

Sample output is generalized as follows:

```
## The following will appear for each trigger program called
TCL BEGIN
TCL: eval {
<Program Internals>
}
## MQL calls in that program and the time taken to execute
## For example
MQL: get env EVENT
MQL time: 0
MQL: get env 0
MQL time: 0
MQL: list attribute
MQL time: 0.01
MQL: set env global mxWzrdListFromCommand.tcl MX-ATTR-1 MX-ATTR-2
MX-ATTR-3 Originator {}
MQL time: 0
TCL time: 0.22 (0.01 in MQL)
TCL END
```

Program Strategies

The following sections provide information on strategies which may be useful to programmers. Sections on helpful features and commands are also included.

Nesting Programs

Programs can be nested—one routine can call another. In this way, programs can be fairly generic. For example, a larger program may need to have an application opened, so it could call a program originally created as a format command.

You must follow these rules when nesting programs:

- External programs can call either MQL or other external programs.
- MQL programs can call other MQL programs only. (With the inclusion of Tcl in MQL, most functions could be written in MQL.)
- Nesting update transactions within a read transaction should be avoided.

You cannot pass eMatrix-defined macros (as listed in the *Macros Appendix*) from program to program. These values from the database can be passed only to the first program called. If you need to pass values between programs, use the RPE, as explained in the section *Using the Runtime Program Environment*.

Notifying Users (Notice, Warning and Error Commands)

MQL contains three commands which allow the programmer to provide graphical messages to the end user. The commands are:

```
notice MESSAGE;  
warning MESSAGE;  
error MESSAGE;
```

where MESSAGE is the text that is displayed to the end user.

These commands pop up a dialog box containing a message and an icon appropriate to the severity level. The advantage of using these commands is that it does not require that you write a Tk interface to report these messages. Some examples:

```
mql notice "Missing value for attribute $szAttr"  
mql warning "Missing value for attribute $szAttr"  
mql error "Missing value for attribute $szAttr"
```

The only difference in the above commands is the severity level. None of them will halt a transaction, but simply provide feedback to the user.

Notice, Warning, and Error commands do NOT halt a transaction.

Use of the percent character (%) in Notice commands may give unpredictable results. If you must use the % character, use two percent signs (%%) so that subsequent characters will be treated as a literal string.

Writing Piped Program Code

Piped programs are run as a separate process, but the launching thread in the core acts as an "mql command" service. Assuming the piped program uses this service as the means for performing its eMatrix-related work (the externally-run program will use the pipes (stdin/stdout) provided to communicate to eMatrix), the piped program is essentially running inside the transaction of the launching thread. Not only does this mean that the piped program can rollback the transaction, it is already authenticated and acting in the context of the launching thread.

You can use the Web client (or ADK) to launch a piped program, which will be run on the server. In this sense, piped programs could be said to work "from the Web." You can use piped programs to implement all your business object methods, perhaps resulting in a "thinner" client.

Piped-programs cannot be downloadable. Therefore, a piped program would be the wrong choice to handle cases where client-side activities need to be performed (for example, a user interface). There are currently no plans to support downloaded piped programs in this general sense.

When external program objects are created, the code section indicates the path to the external program file that should be executed. The external file is either a compiled program or a script file. If a script file is used, it needs to be launched through its interpreter in the defining program object. For example the following are valid code values for piped program objects:

```
c:\ematrix\bin\winnt\PIPEDPROGRAM.exe  
java -cp \ematrix\lib\java PIPEDPROGRAM  
wish80 PIPEDPROG.tcl
```

In the source code of these programs MQL commands can be passed through stdout and the results of these commands retrieved through stdin. When piped program objects are launched, a service thread is started which supports a simple MQL command line interface. The service is available through a full duplex "pipe" mechanism. Programs may issue MQL commands by simply writing the command string to their standard output device, prefixed by an "escape" (\033) character. The result of the command may be fetched by the program by reading the standard input device.

Standard input should be read either a character at a time or a line at a time. Reading the full buffer will seem to hang the system since no end of file mark is generated. The last line of the MQL output stream *for each command* is marked by a new line consisting of a single escape (\033) character, followed by another new line. This means that even if a command has no output, a line containing only the escape character is added to stdin, and will only be cleared by fetching the results.

If it is necessary to terminate an external program, it will not affect eMatrix or whatever process executed the external program.

The sections below show what changes to existing code should be made to take advantage of the pipe service.

Tcl/Tk

When changing an external Tcl/Tk program object to piped, the “mql” is no longer required before MQL commands. You could create a Tcl procedure to write output to `stdout` and read from `stdin` such as:

```
proc mqlCommand {command} {
    puts "\033 $command"
    sets result{}
    gets stdin temp
    while {[string index $temp 0] != "\033"} {
        lappend result $temp
        gets stdin temp
    }
    return $result
}
```

Then, in existing code, whenever an MQL command is issued, replace the preceding “mql” in all MQL lines with `mqlCommand`. For example:

```
mqlCommand "print context"
```

C++

MQL commands should be printed to `stdout`, and read from `stdin`. Sample code:

```
/*
** Include system header files.
*/
#include <stdio.h>
#include <stdlib.h>

/*
** Define global constants.
*/
#define MAX_BUFFER 8192

#define C_MQL_ESCAPE  '\033'
#define C_NEWLINE      '\n'

/*
** Define functions.
*/
char* mql( char* sCommand, char* sOutput );

int main( int argc, char* argv[], char* envp[] ) {

    char sOutput[MAX_BUFFER];

    /*
    ** Example that returns output of multiple commands.
    */
    mql( "print context;list vault;list user;", sOutput );
    printf( "Output 1:\n%s\n", sOutput );

    /*
    ** Example that returns output of a single command.
    */
}
```

```

mql( "print person creator select name business system dump |;", sOutput );
printf( "Output 2:\n%s\n", sOutput );

/*
** Example that returns an error.
*/
mql( "lsit vault;", sOutput );
printf( "Output 3:\n%s\n", sOutput );

/*
** Return to the calling program.
*/
return(0);

}

char* mql( char* sCommand, char* sOutput ) {

/*
** Define local variables.
*/
char* sTmp = sOutput;
char c = (char) NULL;

/*
** Execute the MQL command.
*/
fprintf( stdout, "%c%s%c", C_MQL_ESCAPE, sCommand, C_NEWLINE );
fflush( stdout );

/*
** Get the result.
*/
while ( (c=getc(stdin)) != C_MQL_ESCAPE )
{
*(sTmp++) = c;
}
*(sTmp++) = (char) NULL;
fflush( stdin );

/*
** Return the output.
*/
return( sOutput );
}

```

Java

Similar to C++. Some sample code:

```

import java.io.*;

class test {
    public static void reader() {
        DataInputStream in = new DataInputStream(System.in);

        try {
            String str;

```

```

        do {
            str = in.readLine();
            System.out.println(str);
        }
        while (str.charAt(0) != '\033');
    }
    catch (java.io.IOException e) {
    }
}

public static void main(String[] args) {
    System.out.println("\033 print context;");
    reader();
    System.out.println("\033 print bus Assembly 'MTC 12345' A;");
    reader();
}
}

```

Program Object Select Expressions

Select expressions are used to obtain information about business or administrative objects, which can then be saved as a variable and used in a program. The table below describes the select expressions for Programs and Wizards. For additional information and a complete list of Select expressions, refer to *Appendix A, Select Expressions* in the *eMatrix Navigator Guide*.

Each select expression has a default behavior to minimize the amount of typing required for an answer. In the case of simple data types (string, integer, etc.), the default is always to return the value. Objects have different behavior in each case.

Selectable Fields for Programs/Wizards			
Field	Data Type	Description	Output
classname	string	For Java programs only, the classname of the code.	The classname of the code.
code	string	Program code	For MQL or Tcl, lists the actual code; for external program, lists the command to launch the program.
description	string	Program description	value
doesneedcontext	boolean	Does this Program require a business object?	True/False
doesuseinterface	boolean	Is this Program defined as launchable from an eMatrix Collaboration Server application? (deferred and downloadable)	True/False
downloadable	boolean	Does program include code for operations not supported on the Web product?	True/False
execute	string	How is the execute flag set?	deferred or immediate
hidden	boolean	Is this Program object hidden?	True/False
id	Internal object	Program identifier	value

Selectable Fields for Programs/Wizards			
Field	Data Type	Description	Output
isafunction	boolean	Is program used as a function? (trigger, state check/action, format edit/view/print, frame prologue/epilogue, widget load/validate, button command)	True/False
isamethod	boolean	Is this Program associated with a type?	True/False
isjavaprogram	boolean	Is this Program type Java?	True/False
ismqlprogram	boolean	Is this Program type MQL?	True/False
ispipedprogram	boolean	Does the program use the piped service?	True/False
iswizardprogram	boolean	Is it a wizard program?	True/False
modified	time-stamp	time and date the program/wizard was last modified	value
name	string	Program name	value
originated	time-stamp	time and date the program/wizard was created	value
property[].*	Admin. object	Associated properties	If property[] is specified, returns property names and values. If blank, returns all property names and values.
type	Admin. object	Includes all types which are derived from types specifying the program as a method (that is, all types for which the program is an inherited method)	Type for which the specified program is a method.

The system attempts to produce output for each select clause input, even if the object does not have a value for it. In this case, an empty field is output.

Program With Arguments

The keywords `method` and `program` have been added as select keywords for business objects. They can be used wherever select keywords for business objects can be used, so they can be used in both `print bus` and in expressions evaluated against business objects. The syntax is:

```
program [PROGRAM_NAME ARG1 ARG2 ...]
```

Or:

```
method [METHOD_NAME ARG1 ARG2 ...]
```

where `PROGRAM_NAME` is the name of a program

where `METHOD_NAME` is the name of a method

Zero or more arguments can be included as indicated. The total number of characters in brackets is limited to 127.

The `program` keyword allows non-method programs to be run.

- They can use the same macros as a method.
- The arguments are space-delimited.
- The program/method name must be quoted if it contains spaces.
- Arguments containing spaces must be quoted or enclosed in {}.
- If the program exits with a non-zero exit code, the select clause returns an empty string as a value.

The select clause returns a string whose value is specified by the program. The program specifies it by placing the string in the GLOBAL RPE variable whose name equals the program name. This is consistent with the use of a program to load a single valued text widget, except for having to use `global`.

Note that as a select clause, this capability can be used in any of the following ways:

- `print bus T N R select owner current program[NAME] attribute[ATT] dump`
- to load Tcl variables via: `set svar [mq1 print bus]`
- as part of a where clause in a Query or Cue
- as the expression to be provided in a Tip
- as a column definition

The use of programs as column headers should be done only in special circumstances, since it will significantly slow down table display because the program has to be run for each row.

Example:

For the program ‘Program 27’ (code shown below), the command:

```
print bus Bug 202740 0 select program[‘Program 27’ a b c]
```

generates output as follows

```
Run by the owner of Bug 202740 0:
    business object Bug 202740 0
        program[‘Program 27’ a b c] = You own this object
Run by a non-owner :
    business object Bug 202740 0
        program[‘Program 27’ a b c] = You do NOT own this
object
Run by a guest:
    business object Bug 202740 0
        program[‘Program 27’ a b c] = You are guest!!
```

The following is the code for ‘Program 27’:

```
tcl;
#
# A local procedure to allow turning debugging puts on or off
# by setting/unsetting the rpe variable MXDEBUGFLAG
proc DEBUG { str } {
    if {[mq1 get env global MXDEBUGFLAG] > 0} {puts $str}
}
eval {
```

```

# Get the current user, and type,name,rev
set sUser [mql get env USER]
set sType [mql get env TYPE]
set sName [mql get env NAME]
set sRev [mql get env REVISION]

# Get the program name into which the return string will be written.
set progname [mql get env 0]
set exitCode 0

# Verify that arguments are passed in
for { set indx 1 } { $indx < 10 } { incr indx } {
    set var$indx [mql get env $indx]
    DEBUG "var$indx=[mql get env $indx]"
}

# Require at least one argument
if { $var1 == "" } {
    set exitCode 1
}

# Get owner
if { $exitCode == 0 } {
    set exitCode [catch {mql print bus $sType $sName $sRev \
        select owner dump} sOwner]
}

# Set a return value.
if { $exitCode == 0 } {
    if { $sUser == "guest" } {
        set retval "You are guest!!"
    } elseif { $sOwner == $sUser } {
        set retval "You own this object"
    } else {
        set retval "You do NOT own this object"
    }
}

# Write the return value to the GLOBAL program RPE variable
DEBUG "Return in progname: $retval"
mql set env global $progname $retval
}

DEBUG "ExitCode: $exitCode"
exit $exitCode
}

```

Programming for Use in a Loosely-Coupled Database

When writing program code that will be used in Loosely-Coupled Database (LCD) environments, be sure to use the following guidelines:

- When querying the database, * in the Vault field searches local vaults only. Be specific when you need to find objects in the remote vault.
- When printing objects, use object IDs instead of Type, Name, and Revision. Type, Name, and Revision may not be unique in this environment. Alternatively, you can use the `in vault` clause to unambiguously specify the object you wish to use.

Programs Exported to XML

When programs are exported to XML format, the code is enclosed in a CDATA tag as follows:

```
<CDATA[ PROGRAM CODE... ]>
```

Do not use the end mark “]]>” in your program code. If you have used it in your programs for something similar to “[attr[Cost]]>100”, you should add a space between the bracket and the greater than sign. This allows the program to be both exported and executed properly.

Programs Using Output Length

Due to the addition of selectables pertaining to the creation and modification dates of administrative objects, Tcl programs that use print commands and parse the information based on a certain length of the output will not retrieve the desired information.

For example, the following code will not retrieve the expected value:

```
set x [split [mql print type ASSEMBLY]]  
set xLength [llength $x]  
set value [lindex $x [expr $xLength - 5]]
```

Two lines have been added to the output for creation and modification dates, so the last line above should now be:

```
set value [lindex $x [expr $xLength - 7]]
```

Programs should be written to use specific selectables to avoid this issue. For example:

```
set value [mql print type ASSEMBLY select id dump]
```

Optimizing Programs

This section contains standards and suggestions to help for improving the performance and/or maintenance of custom code.

Access Business Objects and Relationships by Vault or OID

All queries for business objects should use the `vault` keyword if possible. Otherwise, the core must execute an SQL statement for all vaults in the database.

Similarly, if the business object's database identifier (OID) is known, it should be used whenever possible when executing a 'bus' MQL command. If only the type, name, and revision are known, the `in vault` clause should be used to improve performance. The RPE contains variables for both business object and relationship OIDs (OBJECTID and CONNECTID).

For example, this statement:

```
expand bus Part P1 A from relationship EBOM;
```

can be rewritten as:

```
expand bus $objId from relationship EBOM dump terse;
```

This eliminates the need for the system to search every vault before the expand is performed.

"Expand bus Part P1 A ..." starts by searching every vault for Part P1 A until it finds it.

```
select * from lxBO_5f39822c where
    lxType=:va and lxName=:vb and lxRev=:vc
        :va=1737757489
        :vb=P1
        :vc=A
select * from lxBO_678a4321 where
    lxType=:va and lxName=:vb and lxRev=:vc
        :va=1737757489
        :vb=P1
        :vc=A
select * from lxBO_876cdeal where
    lxType=:va and lxName=:vb and lxRev=:vc
        :va=1737757489
        :vb=P1
        :vc=A
```

This search may be repeated in every vault until the object is found.

"Expand bus \$objId ..." skips these preliminary queries altogether. The `terse` clause on the end saves one or more instances of the following:

```
select t.* from lxBO_0a70d4cb t where t.lxOid in (select lxToId
from lxRO_0a70d4cb where lxFromLat=:va and lxFromId=:vb and
lxToLat=:vc)
    :va=175166667
    :vb=1984423433
    :vc=175166667
```

This query is used to get the Type Name Revision of the related objects. It is unneeded with the `terse` clause. The related object's OID is free. This is especially true if you are going to do further processing on the related object. (For example, a further expand or an

update.) The same initial query savings from above apply every time you use an OID instead of TNR.

Selecting Relationship and Business Object Data

```
set [ lResults [ split [ mql temp query bus T N R dump | ] \n ]
foreach sLine $lResults {
    set lBus [ split $sLine | ]
    set sOneData [ mql print bus [ lindex $lBus 0 ] [ lindex $lBus 1 ] \
                  [ lindex $lBus 2 ] select attribute\[ONE\] dump ]
    set sTwoData [ mql print bus [ lindex $lBus 0 ] [ lindex $lBus 1 ] \
                  [ lindex $lBus 2 ] select attribute\[TWO\] dump ]
    #
    # Do something with the data
    #
}
```

Multiple selectable items can be specified for any of the following commands:

- print bus
- expand bus
- temp query
- print set

The coding approach shown above could be made more efficient by using the following syntax:

```
set [ lResults [ split [ mql temp query bus T N R select id \
                        attribute\[ONE\] attribute\[TWO\] dump | ] \n ]
foreach sLine $lResults {
    set lData [ split $sLine | ]
    set sOID [ lindex $lData 3 ]
    set sOneData [ lindex $lData 4 ]
    set sTwoData [ lindex $lData 5 ]

    #
    # Do something with the data. Note that the OID has also been retrieved
    # for all future processing.
    #
}
```

When an MQL selectable item is used to retrieve information about a relationship, the “relationship” item should not be used if a direction is known. Instead, “to” or “from” should be used. This reduces the number of database tables that need to be searched. For example, instead of:

```
relationship\[BOM\].attribute\[Qty\]
```

Use:

```
from\[BOM\].attribute\[Qty\]
```

Referencing Schema Object Names

Avoid hard coding names of schema objects. Instead, use a strategy to look up the names of schema objects such as that found in the Application Exchange Framework or the Application Library.

Query Performance

The key consideration for optimizing query performance is to use as many indexed fields as possible. This limits the number of business objects returned to the client in order to perform the non-indexed evaluation. It's important to limit the use of wildcards for type, name, and revision, and to specify a search vault.

Never use a temporary query to print information about a business object if the type, name, and revision of the business object are already known. For instance, the following code fragment should never exist:

```
set sType [ mql get env TYPE ]
set sName [ mql get env NAME ]
set sRev [ mql get env REVISION ]
set lTmp [ split [ mql temp query bus $sType $sName $sRev \
    select current dump | ] | ]
set sStatus [ lindex $lTmp 3 ]
```

Use a `print bus` command instead. Ideally, this command would use the business object ID instead of the type, name, and revision as discussed in [Access Business Objects and Relationships by Vault or OID](#).

One reason that a programmer may be tempted to use a query to print business object data is because it will not cause an error if the type, name, and revision does not exist. Even in these cases, there are better ways to write the code:

- Use the Tcl `catch` statement around the `print bus` commands to catch errors.
- Execute a `print bus` command with the `exists` selectable item to first see if the business object exists, then issue a second `print bus` command to get the required data. While this approach requires two MQL commands instead of one, the resulting SQL is typically more efficient than the SQL generated from the `temp query` command.

Avoid Initializing multiple Tcl Interpreters

Due to the time to initialize the Tcl session, a performance penalty is incurred every time an MQL/Tcl program is executed. For information on reducing this penalty using the Tcl Interpreter Pool feature, see [Pooled Clause](#). Also, programmers should be aware of the following:

- Use wizard frame prologue and epilogue programs instead of widget load and validate programs if the frame has many widgets.
- Avoid using the `exec program` or `execute businessobject` commands (or their ADK equivalents) to execute reusable code. A better approach is to define the reusable code as MQL/Tcl library program objects.

For systems running the CORBA Collaboration Server on UNIX, use the `MQL shell` command instead of the `Tcl exec` command. Due to a known bug in IONA Orbix, the `Tcl exec` command causes the server and/or the Orbix Daemon (`orbixd`) to crash. Also, if an `edit` or `view` command calls an external program, there could be random failures in eMatrix versions prior to 8.5.1.1 and 9.0.1.0.

- Only use program expressions in an eMatrix table if absolutely necessary. Consider making minor schema changes if it eliminates the need to execute an MQL/Tcl program to populate the cell for each business object row.

Using Transaction Boundaries

Transaction boundaries (`mql start trans` and `commit/abort trans` commands) are primarily used to roll back processing if an error occurs. If many commands are required to successfully complete a logical operation, then transaction boundaries are used. Note that in the case of event triggers, unless they are external or a deferred action, the Matrix or MQL interface layer automatically applies transaction boundaries, and explicitly setting them in the code causes an error.

Using transaction boundaries for methods or wizard code can provide some performance benefits even if the commands inside the boundaries are not a single logical operation. When no transaction is in effect, instance data (business objects, relationships, attributes, etc.) must be reloaded from the database for each MQL command. Transactions can be used to keep data live in the cache between commands, which improves performance.

Storing “Static” Data

Program objects are often used to store information that is seldom changed and is required when executing custom code. In these cases, the program object does not have code in it. It is just being used as a text buffer. Some implementations have used program objects to store the results of queries, which take a long time to execute, but have results that change infrequently. The program object's contents are updated as a triggered event whenever any of the “queried” data is created, modified, or deleted. This approach allows the programs that need the query results to access the program object's contents instead of performing a time-consuming query. Basically, this approach penalizes the user who makes the infrequent change, but rewards the users who make the frequent queries.

The RPE is also an excellent technique to store this kind of information. With the RPE, the initial MQL commands to retrieve the data must still be executed. However, the results can then be stored in the RPE to avoid re-executing the same commands later in the session. It is quite common to store this information in the RPE during the “ValidatePasswordAction” program since end users are typically more tolerant of slower performance when logging onto the system.

The techniques described in this section are very valuable for improving the performance of a Wizard's widget load programs if they currently execute a query.

Program Exit vs. Program Failure

After eMatrix executes a program, it will first determine if the program executed successfully or failed. If it failed, eMatrix will raise an exception, which generally aborts the transaction and displays an error. If it executed successfully, in some cases eMatrix will then look at the exit code returned by the program. The table below shows the eight cases where the exit code of a program is evaluated, and what a non-zero exit code means to eMatrix. The effect of failures is also listed.

Program Type	Non-zero Exit Code means...	Failure causes...
state action	to block the promotion of a business object and display "Can't promote T N R" message.	the promote transaction to be aborted, and an error to be displayed; essentially same as a block with non-zero exit code.
state check	to block the promotion of a business object and display "Not all requirements satisfied" message.	the promote transaction to be aborted, and an error to be displayed; essentially same as a block with non-zero exit code.
trigger check	to block the event and display an error.	the event transaction to be aborted, and an error to be displayed; essentially same as a block with non-zero exit code.
trigger override	to skip the event, but commit the transaction.	the event transaction to be aborted.
wizard validate	to remain on the current frame.	the Wizard to fail.
wizard epilogue	to remain on the current frame.	the Wizard to fail.
wizard prologue	to skip the current frame.	the Wizard to fail.
wizard button	to redisplay the current frame.	the Wizard to fail.
attribute range	that the entered value is not acceptable, and to display an error. The exit code is only evaluated for this type of program when EVENT = "attribute check." Refer to Programs for Attribute Ranges for more information.	the values provided by the program to be unavailable for selection or validation. The only values that can be written to the attribute are those that are within other defined ranges.

Note that for both state checks and trigger checks, the outcome of a failure is the same as when the exit code is non-zero. So for these two cases, programmers could accomplish the same thing either by designing the program to fail or by returning a non-zero exit code.

In general, exit code is *not* checked when eMatrix is returned to the idle state after a program is executed. This occurs in the following circumstances:

- Programs executed as Action Triggers
- Programs executed as methods
- Programs executed from a toolbar
- Wizard body code
- When any program fails

This is not to say that a method, for example, that contains nested programs cannot pass exit codes between those routines.

Note that lifecycle actions are not on this list. Exit code IS evaluated on lifecycle actions, where a non-zero exit code will abort the promote transaction.

Generating an Exit Code

In general, exit codes are evaluated by eMatrix only when the program is executed within a wizard or trigger (or lifecycle check or action), where the flow of control keeps moving (that is, eMatrix still has more to do, and is not idle). Exit codes are generated in a variety of ways, depending on the platform and programming language being used.

To return an exit code

- From an internal (MQL) program in MQL mode, use the `quit` command. The `quit` command passes a value back through the use of an integer argument (for example, `quit 0`).
- From an internal (MQL) program in Tcl mode, use the `exit` command. The `exit` command passes a value back through the use of an integer argument (for example, `exit 0`).
- From an external program, use the normal exit code mechanism supported on each platform and programming language.

Note that the Tcl command “`return 1`” is NOT evaluated by eMatrix as an exit code.

Exit Inside of an Eval Statement

An exit inside of an eval statement is processed differently than you might expect.

eMatrix replaces the real Tcl exit command with its own in order to prevent an exit command from shutting down MQL, eMatrix, Collaboration Server, etc. as well. eMatrix sets a flag, which it checks before passing a new statement to Tcl. However, all that eMatrix sees of the statement in the following example is `eval { ... }`, just one big eval statement. eMatrix must exit before it sees that the flag was set.

So the following code continues to execute after the `exit` command:

```
tcl;
eval {
    exit
    puts "This should never print."
}
puts "Tcl will exit before running this."
The following code works as would be expected.
tcl;
eval {
    exit
```

```
    return
    puts "This should never print."
}
puts "tcl will exit before running this."
```

The built in return forces it to stop executing the eval statement, which then notices that the exit flag was set and closes Tcl.

Program Failure

Programs will fail in certain circumstances. Syntax errors are obvious causes of program failure. System errors caused by MQL that makes assumptions (such as the existence of a business object or relationship), also fail, displaying an error message and aborting the enclosing transaction whether or not it is caught and handled by the tcl "catch" command. Therefore, always avoid using MQL commands which could generate system errors. For example, instead of:

```
tcl;
eval {
    if {[catch {mql print relationship doesnotexist} err]} {
        puts "does not exist"
        exit 0
    }
}
```

check for the existence of the relationship like this:

```
tcl;
eval {
    if {[mql list relationship doesnotexist] == ""} {
        puts "does not exist"
        exit 0
    }
}
```

Programmers can also explicitly fail a Tcl program by using the native `error` command. But no matter how a program fails, be aware that the exit code is not checked by eMatrix since the transaction was aborted.

Note that use of the mql error command will NOT cause program failure. Refer to [Notifying Users \(Notice, Warning and Error Commands\)](#) for more information.

Dialog Scripting Language

Programs can be written that can call many eMatrix dialogs through the use of the application command. The application command provides programmers with a graphical user interface alternative to Tk, and one that provides eMatrix familiarity. Since the dialogs must be accessible from the launching process, these programs can be:

- An MQL or Tcl program object run either within eMatrix (via a tool, method, or trigger) or with the “matrix -mql” command line option. (matrix -mql -c “execute program NAME;”).
- An MQL script run with the “matrix -mql” command line option (matrix -mql -c “run FILE.mql;”).
- Commands passed by an external program or application that is talking to matrix -mql through an MQLIO connection that was opened using the mqlopen2 interface, passing matrix.exe as the ‘executable’ argument.

Refer to [Command Line Execution](#) for more information on the first two options. MQLIO is documented in *Embedding MQL*.

Special considerations must be made when writing code for programs that will be run on the Web. See [Dialog Scripting on the Web](#) for details.

Application Command

The application command is used by specifying the dialog type and the full specification of a business object, if required. The syntax is:

```
appl DIALOGTYPE [bus OBJECTID] [program PROGRAM_NAME {ARG}];
```

where:

OBJECTID is the Type Name Revision of the business object.

PROGRAM_NAME allows a program to be launched when the application dialog is dismissed. A dialog is dismissed when a user clicks the Close or Cancel button, a dialog-specific commit button (such as Modify or Checkin), or the X in the upper right corner of the dialog. This “dismiss program” can be used, for example, to allow a wizard sequence to be customized to include an application dialog. Any variables that need to be maintained can be passed to the specified program as arguments. When a program is launched in this manner, the INVOCATION macro is set to “program.” (See [Invoking a Wizard from a Program](#) in Chapter 18.)

ARG is zero or more space-delimited strings that are passed to the program as arguments.

The program clause with all its arguments must be at the end of the appl command. Everything after the program name is assumed to be arguments to the program.

DIALOGTYPE can be:

attributes	basic	change
checkin	checkout	clone
details****	edit***	files
find*	formats	forms
history	icons****	indented

indentedtable	methods	navigator
notify**	paginatedtable	print***
reassign	relationshipattributes	relationshiphistory
revision	revisions	route*
sendmail	star	states
view***	wizard	.

A few points need to be made about some of the DIALOGTYPES:

- * Does not require that a business object is specified.
- ** Notify is not a dialog type, but is used with the application command to update an appl dialog's visuals when a business object has been modified. See [Updating Browsers](#).
- *** A format and/or a file can also be specified when appl is used with edit, print or view, as follows:

```
appl | edit | bus OBJECTID [format FORMAT [file FILENAME]];
      | print |
      | view |
```

When any of these forms of the appl command is used, an eMatrix dialog is not displayed, but the format's edit, view or print program is launched.

- ****The details and icons dialog types let you add objects or sets to eMatrix browsers in Details or Icon mode. See [Adding an Object or a Set to Specified Browser](#) for additional options.

DIALOG values are CASE SENSITIVE and can NOT be abbreviated.

For example, the following command could be used to call the attributes dialog for whatever object the program passes:

```
appl attributes bus OBJECTID
```

Additional Functionality

The application command provides several additional functions:

Adding an Object or a Set to Specified Browser

An object or set can be added to an eMatrix primary browser, in either Icons or Details mode. In addition, several options exist for the programmer to specify in which primary browser they should be added.

To add an object or set to a primary browser, the following commands are available:

appl[ication] icons bus OBJECTID [dialog WHICHDIALOG];	
details set SETNAME	

where:

OBJECTID is the OID or Type Name Revision of the business object.

SETNAME is the name of a valid set in the context user's workspace.

WHICHDIALOG indicates the primary browser to which object(s) will be added. It can have one of the following values:

- **first**—the first primary browser that was opened in the session
- **all**—all existing primary browsers
- **DIALOGID**— the primary browser that was opened through a previous application scripting command and returned the specified DIALOGID to the calling program
- **new**—meaning a new primary browser will be opened and loaded with the specified object(s)

If the dialog WHICHDIALOG clause is not used, the specified object or set will use the default and be loaded into a new primary browser. When a particular dialog is specified, its mode (icons/details) can be changed with this command.

For example, to add MySet to all primary browsers and switch them all to Details mode, use:

```
appl details set MySet dialog all;
```

To add the Release Notes Manual object to a new primary browser use:

```
appl icons bus Manual "Release Notes" 1 dialog;
```

Specifying a View

The appl navigator commands provide the option to specify a view, allowing a set of visuals (tips, filters, cues, and a default table) to be enabled. The following commands can optionally include a view specification:

```
appl indented bus OBJECTID [view VIEWNAME]
appl indentedtable bus OBJECTID [view VIEWNAME]
appl star bus OBJECTID [view VIEWNAME]
appl navigator bus OBJECTID [view VIEWNAME] (This one looks at the user's "browse by" preference to display preferred relationship browser.)
```

Where:

OBJECTID is the OID or Type Name Revision of the business object.

VIEWNAME is the name of a pre-defined view. Performance is optimized since the view is applied before the object is expanded.

Updating Browsers

When modifications are made to an object from either the Attributes or Change dialogs, or from a form, all visuals active in any existing Primary or Navigator browsers that contain that object are updated automatically. Object state changes are also tracked and updated.

In addition, when these types of changes are made via a program (method, trigger, or wizard), the following command can be incorporated into that program to reapply the visuals:

```
appl notify bus OBJECTID modified;
```

Similarly, the following command can be used to update browsers to reflect newly-created objects:

```
appl notify bus OBJECTID created;
```

The notification of such change is sent only to the dialogs/browsers that are controlled by the SAME process as the one doing the notifying. For example, if you launch one eMatrix Navigator session (Session1), and another eMatrix Navigator session (Session2), the

appl notify command typed into Session2 would NOT affect any of the dialogs/browsers of Session1.

Checkin Parameters

It is possible to hide some of the check boxes on the checkin dialog and set the value to a specific value. For example:

```
appl checkin flags FLAG_NAME;
```

FLAG_NAME can be any of the following:

- append or replace
- unlock or notunlock
- delete or notdelete

Checking Out Sets

Entire sets can be checked out with the following command:

```
appl checkout set NAME;
```

The checkout dialog is displayed, listing the objects contained in the named set. The user can then check out the files contained in all objects in the set.

Displaying Forms

Specific forms can be displayed for the passed business object using the following command:

```
appl forms bus ${OBJECT} [FORM];
```

If a FORM is specified, it is displayed against the object passed. Without a named form, the form chooser is presented to the user for a selection.

Reassigning a Set

The contents of an entire set can be reassigned to a single user by popping up the reassign dialog:

```
appl reassign set NAME;
```

The reassign dialog is displayed listing all objects contained in the set.

Displaying Relationship Attributes and History

The attributes dialog of a connection can be displayed using the following application command:

```
appl relationshipattributes CONNECTIONID;
```

where CONNECTIONID is the identification number of the connection.

This is helpful when you are programmatically connecting objects, and you want to get user input for the attribute values.

The history dialog of a connection can be displayed using the following application command:

```
appl relationshiphistory CONNECTIONID;
```

where CONNECTIONID is the identification number of the connection.

Sending a Set

Entire sets can be sent via IconMail using the following command:

```
appl sendmail set NAME
```

The IconMail dialog is displayed with all objects contained in the named set listed. The message is sent once, but contains all members of the named set.

Application Help

Help on the syntax of the application command is available, but *not* through the MQL `help application` command. Since `application` is not really an MQL command, only being available when eMatrix is run in MQL mode, you must be in this mode to get help as well.

To get help, start eMatrix with the `-mql` argument. On Windows machines, this means modifying the properties of the matrix.exe shortcut. Add `-mql` to the target area of the shortcut tab. Start the application and then type:

```
appl help;
```

Using the Application Command

When the `application` command is passed, the dialog is displayed on the user's screen and the dialog ID# is returned to the calling process. By default, dialogs are invoked non-modally, which means that the calling program continues to operate whether or not it received input from the user. This might be useful, for example, if the calling program wanted to give the user the opportunity to update attributes by presenting the attributes dialog, without needing to know what changes the user made. In this case the calling program could use `appl attributes bus OBJECTID` to cause the attributes dialog to pop up. Control would return immediately to the calling program, which could continue, ignorant of whether or not the user edited the attributes or just cancelled the attributes dialog.

Wait and Modal Clauses

In other cases, it is important for a calling program to know what modifications the user may have made in eMatrix through an `appl` dialog. In these cases, the calling program should follow the dialog invocation with a call to `appl wait`. This command causes execution of the program to pause until the user has disposed of the specified dialog (or all dialogs, for `appl wait 0`). Once the user does that, execution of the program resumes at the statement just following the `appl wait` command. Note that the `appl wait` command does not freeze eMatrix. A user can still use eMatrix even though they have not disposed of the invoked dialog.

The `wait` keyword causes the calling program to be suspended until the specified dialog has been closed. The syntax is:

```
appl wait DIALOGID;
```

where `DIALOGID` is the number output by the `appl DIALOGTYPE` command which popped up the dialog that you want to wait for.

To have the eMatrix application freeze (be unusable) until the user has finished with the invoked dialog, the calling program should follow the dialog invocation with a call to

`appl modal`. For this reason, dialogs that have child windows should not be made modal.

When the `modal` clause is used, the specified dialog becomes modal. Modal dialogs stay in the foreground until they are closed.

```
appl modal DIALOGID;
```

where `DIALOGID` is the number output by the `appl DIALOGTYPE` command which popped up the dialog that you want to be modal.

Note that `appl modal` is intended to be used only with a subset of the `appl` commands. Specifically, it should NOT be used with any dialogs which themselves can invoke additional dialogs. For example, the Find dialog has a button to get to the Query dialog and the Formats dialog has menus for file access, edit, view, etc.; these dialogs should not be invoked in a modal state.

Appl modal is intended to be used ONLY with standalone dialogs. It is not supported on eMatrixApplet.

The following table identifies which dialogs can (OK) or cannot (NO) be used in conjunction with `appl modal`:

<code>appl attributes bus Assemby 12345 1</code>	OK
<code>appl basic bus Assemby 12345 1</code>	OK
<code>appl change bus Assemby 12345 1</code>	NO
<code>appl checkin bus Assemby 12345 1</code>	OK
<code>appl checkout bus Assemby 12345 1</code>	OK
<code>appl checkout set Assemblies</code>	OK
<code>appl clone bus Assemby 12345 1</code>	OK
<code>appl details bus Assemby 12345 1</code>	OK
<code>appl edit bus Assemby 12345 1</code>	OK
<code>appl edit bus Assemby 12345 1 format Word</code>	OK
<code>appl edit bus Assemby 12345 1 format Word file nospace.doc</code>	OK
<code>appl files bus Assemby 12345 1</code>	OK
<code>appl find</code>	NO
<code>appl formats bus Assemby 12345 1</code>	NO
<code>appl forms bus Assemby 12345 1</code>	NO
<code>appl history bus Assemby 12345 1</code>	NO
<code>appl icons bus Assemby 12345 1</code>	OK
<code>appl indented bus Assemby 12345 1</code>	NO
<code>appl indentedtable bus Assemby 12345 1</code>	NO
<code>appl methods bus Assemby 12345 1</code>	NO
<code>appl navigator bus Assemby 12345 1</code>	OK
<code>appl print bus Assemby 12345 1</code>	OK
<code>appl print bus Assemby 12345 1 format Word</code>	OK

appl print bus Assemby 12345 1 format Word file nospace.doc	OK
appl reassign bus Assemby 12345 1	OK
appl reassign set app	OK
appl relationshipattributes connection 19.24.5.16	OK
appl revision bus Assemby 12345 1	OK
appl revisions bus Assemby 12345 1	NO
appl route bus Assemby 12345 1	OK
appl sendmail bus Assemby 12345 1	OK
appl sendmail set app	OK
appl star bus Assemby 12345 1	NO
appl states bus Assemby 12345 1	NO
appl view bus Assemby 12345 1	OK
appl view bus Assemby 12345 1 format Word	OK
appl view bus Assemby 12345 1 format Word file nospace.doc	OK

Appl modal is not supported for the Web version of eMatrix.

The difference between `appl wait` and `appl modal` is only evident when running a Tcl/MQL script in an interactive `matrix -mql` session, in which case `appl wait` allows the user to continue to type ahead in the `matrix -mql` text window; `appl modal` does not.

Output

Most dialogs output only the ID to the calling program, and this is done as soon as the window is opened. User input may affect the eMatrix database, but is generally not explicitly fed to the calling program. For example, if the attributes dialog is popped up and the user changes a value and clicks the edit button, the database is updated, but the new values are not automatically passed to the program. If the program needs to know the new values, it still has the object ID, and can execute its own MQL commands to retrieve them. However, with dialogs like change, clone, and revision, the program could lose the handle to the object if the type, name or revision is changed. In these cases, the object type, name, and revision are output to the program.

Programming for the Web Version

When writing programs that may be used with eMatrixApplet (Web version of eMatrix Navigator), there are considerations that need to be addressed. This section provides some tips to keep in mind if programs will be run on the Web.

Downloadable Programs

In order for a program to be executed on the Web client, it must be marked as Downloadable (see [Downloadable Clause](#)). Downloadable programs can be run only in a deferred mode. Therefore, selecting the **Downloadable** option automatically sets the Execute option to Deferred (see [Execute Clause](#)).

If Downloadable is specified and the Execute option is Immediate, an error is generated. Likewise, if on program modification, a command mismatch occurs, an error is generated that reads:

A program that is downloaded cannot execute immediately.

Programs not marked downloadable that are executed via a Web session (a trigger, for example) are executed on the Collaboration Server machine.

Downloadable Program Threads Share Context

Beginning with the version 9 series of the eMatrix products, the handling of downloaded programs has been modified to fit within a “Web-centric” model. For technical reasons, all threads running on behalf of the Web browser share a single context. In versions prior to 9, each thread had its own independent context. This change has these two important implications, which are described in detail in the following sections:

- eMatrixApplet runs in its own thread and each downloaded program runs in its own thread. To avoid having the common context affected by concurrent threads, the eMatrixApplet thread is suspended until the downloaded program thread has completed. Therefore, all windows and dialog boxes associated with the eMatrixApplet are disabled until the other threads are completed.
- A single Runtime Program Environment (RPE) is associated with the single context that is now shared among all threads. In the rare cases when multiple downloaded programs run concurrently, this shared RPE may cause problems.

Disabled eMatrixApplet Windows

While one (or more) downloaded programs are executing on the client, the eMatrixApplet dialog boxes and windows are disabled. Any dialogs displayed by the downloaded program(s) are fully operational. When all downloaded programs complete, the eMatrixApplet dialog boxes are once again enabled.

A downloaded program is deemed completed when it either exits or fails. The wish80 Tcl interpreter does not treat the Tcl `return` command the same as the Tcl `exit` command. According to the Tcl documentation, a `return` command should only be used within a Tcl procedure or inside a file that is passed to the Tcl `source` command. The only proper way to exit from the main code of a Tcl program is to use the Tcl `exit` command. This is extremely important now since a downloaded program that uses the `return` command where an `exit` command should have been used results in the eMatrixApplet never gaining control back. The only workaround is to kill the orphaned wish80 executable,

which signals that the downloaded program has finished; control then returns to eMatrixApplet.

The exit, return, and source Tcl commands are described in detail below.

Tcl Command	Synopsis	Description
exit	exit ?returnCode?	Terminate the process, returning <code>returnCode</code> to the system as the exit status. If <code>returnCode</code> isn't specified, then it defaults to 0.
return	return ?-code code? ?-errorinfo info? ?-errorcode code? ?string?	<p>Return immediately from the current procedure (or top-level command or source command), with string as the return value. If string is not specified, then an empty string is returned.</p> <p>In the usual case where the <code>-code</code> option isn't specified, the procedure returns normally (its completion code will be <code>TCL_OK</code>). However, the <code>-code</code> option may be used to generate <i>Exceptional Returns</i> from the procedure.</p>
source	source fileName source -rsrc resourceName ?fileName? source -rsrcid resourceId ?fileName?	<p>This command takes the contents of the specified file or resource and passes it to the Tcl interpreter as a text script. The return value from <code>source</code> is the return value of the last command executed in the script. If an error occurs in evaluating the contents of the script, then the <code>source</code> command returns that error. If a <code>return</code> command is invoked from within the script, then the remainder of the file is skipped and the <code>source</code> command returns normally with the result from the <code>return</code> command.</p> <p>The <code>-rsrc</code> and <code>-rsrcid</code> forms of this command are only available on Macintosh computers. These versions of the command let you source a script from a TEXT resource. You may specify what TEXT resource to source by either name or id. By default, Tcl searches all open resource files, which include the current application and any loaded C extensions. Alternatively, you can specify the <code>fileName</code> where the TEXT resource can be found.</p>

Exceptional Returns

The `-code` option is rarely used. It is provided so procedures that implement new control structures can reflect exceptional conditions back to their callers. *Code* may have any of the following values:

- **ok**

Normal return: same as if the option is omitted.

- **error**

Error return: same as if the `error` command were used to terminate the procedure, except for handling of `errorInfo` and `errorCode` variables. These two variables may be used to provide additional information during error returns. They are ignored unless `code` is **error**.

The `-errorinfo` option specifies an initial stack trace for the `errorInfo` variable. If `errorInfo` is not specified, then the stack trace left in `errorInfo` includes the call to the procedure and higher levels on the stack but it does not include any information about the context of the error within the procedure. Typically the `info` value is supplied from the value left in `errorInfo` after a `catch` command trapped an error within the procedure.

If the `-errorcode` option is specified, then `code` provides a value for the `errorCode` variable. If the option is not specified, then `errorCode` defaults to `NONE`.

- **return**
The current procedure returns with a completion code of `TCL_RETURN`, so that the procedure that invoked it returns also.
- **break**
The current procedure returns with a completion code of `TCL_BREAK`, which terminates the innermost nested loop in the code that invoked the current procedure.
- **continue**
The current procedure return with a completion code of `TCL_CONTINUE`, which terminates the current iteration of the innermost nested loop in the code that invoked the current procedure.
- **value**
`Value` must be an integer; it is returned as the completion code for the current procedure.

Shared RPE

Having a single shared context implies a single shared RPE. Therefore all threads read and write to the same RPE. Since eMatrixApplet is disabled during the running of a downloaded program, this is not a problem when a single downloaded program is executed. Similarly, a downloaded program that causes another downloaded program to execute (referred to as *chaining*) is not a problem. However, if multiple downloaded programs are executing concurrently then there are issues to be aware of.

First, since all programs are sharing the same RPE, when multiple downloaded programs are running concurrently there is the potential for one program to overwrite an RPE entry that was defined by another program. This behavior can actually be leveraged for positive gain when performed in a controlled fashion. When this is not desired behavior, it is a good idea to follow a convention for naming RPE entries that minimizes the chance of name clash. For example, since program names must be unique, each program could name its RPE entries by pre-pending the program name to the front of the variable name.

Second, since programs now share the same RPE, there are problems when one program, a parent program, launches multiple downloaded child programs. Unlike “chaining”, where one program runs after the previous program is downloaded, this “multiple downloaded children” case results in all programs being downloaded at the same time (even if there are lots of lines of code in between the `exec` statements that launch each child program). This is due to the fact that downloaded programs are deferred and therefore queued up for execution when the system commits the transaction surrounding the parent program.

Along with each program on the queue is a snapshot of the RPE when the program was to be executed (that is, when the `exec` statement was encountered). This saved RPE snapshot is restored when the program is removed from the queue and executed. Normally (for deferred programs that are not downloaded) this results in the correct behavior, including the fact that any arguments passed to the program are correctly found in the RPE entries “0”, “1”, etc. However, since a number of queued, downloaded programs will be removed from the queue before they actually run (due to the latency of getting down to the client) the result is the snapshot associated with last program on the queue will overwrite the RPE. For example, the arguments passed to the last program will be found in the RPE entries “0”, “1”, etc. and therefore all programs will pick up these values (including the ones queued to execute prior to the last program on queue).

Tcl Programs

If Web client users need to use tools, wizards or methods written with Tcl, they must install Tcl/Tk version 8 and a required eMatrix .dll file on the client machine. For details, see *Setting up eMatrix for Web Client Access* in the *Installing eMatrixApplet* chapter of the *eMatrix Installation Guide*.

File Checkin and Checkout

Two clauses of the checkin and checkout commands are designed for use on the Web only. The **client** and **server** clauses allow programmers to specify where files are to be located when their programs are executed from a Web client (either downloaded or run on the Collaboration Server). These clauses are used to override the defaults and are ignored when executed on the desktop client.

By default, checkout commands executed from the Web land files on the Web client machine. The **server** clause allows the programmer to alternatively specify the Collaboration Server for the operation. For example, the following statement in a Tcl program object that is run from the Web client will land the file text.txt on the client:

```
mql checkout bus Assembly Wheel 0 format ascii file  
\tmp\text.txt;
```

while the following will land it on the server:

```
mql checkout bus Assembly Wheel 0 format ascii server file  
\tmp\text.txt
```

The **client** clause used with the checkin command provides the same alternatives. The default file location for checkin from the Web is the Collaboration Server, so a programmer may specify **client** to have the file copied from the Web client machine instead. For example:

```
mql checkin bus Assembly Wheel 0 format ascii file  
\tmp\text.txt;
```

would look for the file text.txt on the server, while the following would look for it on the client:

```
mql checkin bus Assembly Wheel 0 format ascii client file  
\tmp\text.txt;
```

In order to be consistent, the **client** or **server** clause may be specified with both checkin and checkout. For example:

```
mql checkout bus Assembly Wheel 0 format ascii client file  
\tmp\text.txt;
```

```
mql checkin bus Assembly Wheel 0 format ascii server file  
\tmp\text.txt;
```

will yield the same results as:

```
mql checkout bus Assembly Wheel 0 format ascii file  
\tmp\text.txt;
```

```
mql checkin bus Assembly Wheel 0 format ascii file  
\tmp\text.txt;
```

Dialog Scripting on the Web

Most application commands, used to display eMatrix dialogs, are supported on the Collaboration Server. Therefore, users of eMatrixApplet (Web Navigator) and other applications that use the Collaboration Server to access an eMatrix database can execute these types of programs. For additional information, see [Dialog Scripting Language](#).

All parameters passed to an application command must be quoted, as shown in the command list.

OBJECTID is not yet supported by the application command. Objects must be specified by Type, Name, and Revision.

The following application scripting commands are NOT yet supported on the eMatrix Web Navigator, or by the ADK classes:

- appl edit bus "TYPE" "NAME" "REVISION" (all forms of command)
- appl find
- appl modal DIALOGID
- appl print bus "TYPE" "NAME" "REVISION" (all forms of command)
- appl wait "DIALOGID"
- dialog "DIALOGID" forms of appl details and appl icons commands

If a program that contains these commands is launched from an eMatrixApplet session, an error is displayed.

Web URL Access

A correctly structured URL address can be used to access an eMatrix database via eMatrix Web Navigator. Most eMatrix dialogs and wizards can be launched in this manner. The URL could be received in an email message, and, like other links which are sent via email, it can be clicked to bring up a Web browser. A CGI application (matrixcgi) starts eMatrix Web Navigator in the mode set with the URL parameters (for example, with context set and the specified objects in the browser). This CGI application is installed in the cgi-bin scripts directory of the Web server when the eMatrix Collaboration Server is created.

For details, see the *matrixcgi* section in the *Installing eMatrixApplet* chapter of the *eMatrix Installation Guide*.

Executing a Program

To execute a defined program, use the following statement:

```
exec program PROGRAM_NAME [ -method METHOD_NAME ] [ARGS] [ { -construct ARG} ] ;
```

PROGRAM_NAME is the name of the program to execute.

The other parameters are related to Java Program Objects (JPOs):

ARGS is zero or more space-delimited strings and ARG is a single string. If the method returns an int, then this is assumed to be the exit code from the method (which has meaning to the eMatrix system when used as a trigger program, wizard utility program, etc.). If the method returns an Object, then the exit code is set to zero and the object is serialized and passed back in the program's output stream. Obviously, this serialized object is of no use in the MQL environment, but is very useful in the ADK environment.

The `exec program` command can be used to avoid needing to know the JPO mangled class name. However, the class must live inside a JPO (no search is made outside the eMatrix database).

The `-construct` clause is used to pass arguments to the constructor (Note: if more than one argument needs to be passed to the constructor, the `-construct` clause can be repeated as many times as necessary on a single command). This may be necessary to establish an object's identity. For example, if one had a JPO named "Project" that held business logic for managing business objects of type "Project," then this JPO would undoubtedly be derived from the ADK BusinessObject class.

For details on programming JPOs, see the *Java Program Object Programmer's Guide*.

Command Line Execution

eMatrix provides the functionality to run a program without displaying the entire eMatrix application user interface. This may be useful for users who need to enter information (create objects), but do not need to access eMatrix for anything else. On Windows platforms, a shortcut could be added to the user's desktop by the System Administrator. This shortcut could run an eMatrix program without displaying the main eMatrix dialog. For example:

```
matrix.exe -mql -c "execute program PROGNAME ;"
```

You can execute any valid MQL command(s) in this manner. However, note that if additional command line options are used, do not separate the `-mql` and `-c` arguments.

eMatrix Command Line Options

The following options can be appended to the eMatrix command when it is started from the system command line or Windows desktop icon.

Option	Meaning
-wizard NAME	Directly launch business wizard.
-mql	In general, allows a program to use GUI elements of eMatrix (through the <i>Application Command</i>) while running what is mostly an MQL/Tcl program. May be used without -c for use with mqlio and mqlbos programs.
-c "COMMAND[COMMAND];"	To be used with the -mql option to execute COMMAND(s). Must keep -mql and -c arguments together if other command line options are used.
-v	Verbose option for command line interface.
-k	No abort on error for command line interface.
-p	Use piped command line interface.
-browseby VALUE	Browse objects by icons, or details.
-navigateby VALUE	Navigate objects by star, indented or details.
-filterbar	Toggle filter bar on.
-revisionwarn	Warn when object has higher revision.
-filepath PATH	Default directory for checkin/checkout.
-checkoutlock	Lock files on checkout.
-checkoutoverwrite	Overwrite existing files on checkout.
-checkoutconfirm	Confirm file replace on checkout.
-checkinappend	Append files on checkin.
-checkinunlock	Unlock files on checkin.
-checkindelete	Delete files on checkin.
-checkinconfirm	Confirm file replace on checkin
-laf VALUE	Change the 'look and feel' of the eMatrix graphical user interface (GUI). Possible values are windows, motif, openlook. This can be run on any platform to have the eMatrix GUI mimic that look and feel.
<i>The following options can be used on the command line for the eMatrix or Business or System application.</i>	
-bootfile FILE	Set bootstrap file name.
-open VALUE	Open objects for view or edit.
-viewby VALUE	View objects by icon, image or name.

Dealing With User Context

The context settings provided by the user at login time define what types of accesses that user has to eMatrix. Programs, triggers and wizards may be available to users who do not have appropriate access privileges to run them, so context must be changed within the program and then changed back to the original user's context.

Two MQL commands are available that are useful in scripts that require a temporary change to the session context:

Push Context Command

The `push context` command changes the context to the specified person and places the current context onto a stack so that it can be recovered by a `pop context` command. `Push context` can also be issued with no additional clauses, in which case the current context would be placed on the stack, but the current context would be unchanged. The command syntax is:

```
push context [user NAME] [pass PASSWORD] [vault VAULT];
```

Pop Context Command

The `pop context` command takes the last context from the stack and makes it current. The command syntax is:

```
pop context;
```

This feature is highly useful in scripts where it is necessary to change the current context temporarily. Context is set back to original user after the interim program that changed context ends.

For example, a trigger program may need to switch context to perform some action which is not allowed under the current user context. It must then return to the original context to prohibit invalid access or ownership for subsequent actions.

Initialization File Context Variable

Alternatively, there is an initialization file variable that controls whether context is restored or not at the conclusion of a program. If the `MX_RESTORE_CONTEXT` variable is set to true, the original user's context will be restored after the program/wizard/trigger completes. If set to false, any context changes made by a program/wizard/trigger will remain changed after the program completes.

Program Output

When a Tcl program is run, where does its output go? Since eMatrix can run programs on both the desktop client and the Web, the answer to this question may change depending on where the program is run. For example, if a program is run on the desktop client and its output goes the MQL window, the same program run on the Collaboration Server will send its output to the Collaboration Server window.

In a program run by eMatrix, there are three kinds of output from the program:

- Tcl command echo: every Tcl command echoes its output:

```
% set var ^my value^  
my value  
%
```
- Tcl stdout/stderr: output generated by puts, Tcl error, return, exit
- MQL stdout/stderr: the output generated by eMatrix on execution of an MQL command, such as mql output, mql notice, wizards, etc. should always go exactly where the program specifies: MQL output goes to the MQL window; notice and wizards pop up appropriate dialogs.

The following explains how the types of output listed above are treated in different circumstances. For each of the following, if the output goes to the MQL window, a program run on the Collaboration Server will go to the Collaboration Server window. Otherwise, the output is the same whether the program is run on the desktop client or on the Collaboration Server.

1. Tcl commands that are not run inside of some level of {..} will echo to the MQL window. Enclosing the program in eval { ... } suppresses this.
2. Tcl stdout/stderr always goes to the MQL window.
3. Naked MQL command (no ‘set var’, no ‘catch’):

```
% mql command
```

Both stdout and stderr go to the MQL window.
4. When a Tcl variable is used to receive stdout:

```
% set tclVar [mql command]
```

Stdout goes to tclVar, stderr goes to the MQL window.
5. When catch is used to keep error from popping up MQL window:

```
% set z [catch {mql command}]
```

Error CODE (0 or 1) goes to variable z.
Stdout and stderr disappear. Catch prevents them from going to the MQL window, but there is no variable to receive stdout/stdin.
6. Use catch and direct MQL stdout/err to Tcl variable:

```
% set z [catch {mql command} tclVar]
```

Error CODE (0 or 1) goes to variable z.
Stdout/stderr goes to outStr.
7. Results can be different when the MQL command in 4,5,6 itself calls a program. In the examples below, this is shown as exec prog, but it could also be a method, or any command that causes a trigger to fire.

The Tcl output from such a second level is handled the same way: Tcl stdout/stderr goes to the MQL window. The following table shows where MQL stdout, stderr goes:

Command	MQL stdout and stderr:
<code>mql exec prog SubProg</code>	go to MQL window
<code>set tclVar [mql exec prog SubProg]</code>	stdout goes to tclVar, stderr to MQL window
<code>set z [catch {mql exec prog SubProg}]</code>	disappear
<code>set z [catch {mql exec prog SubProg} tclVar]</code>	go to tclVar

For systems running the CORBA Collaboration Server on UNIX, use the MQL shell command instead of the Tcl exec command. Due to a known bug in IONA Orbix, the Tcl exec command causes the server and/or the Orbix Daemon (orbixd) to crash. Also, if an edit or view command calls an external program, there could be random failures in eMatrix versions prior to 8.5.1.1 and 9.0.1.0.

Copying and/or Modifying a Program Definition

Copying (Cloning) a Program Definition

After a program is defined, you can clone the definition with the Copy Program statement. This statement lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy program SRC_NAME DST_NAME [MOD_ITEM {MOD_ITEM}];
```

SRC_NAME is the name of the program definition (source) to copied.

DST_NAME is the name of the new definition (destination).

MOD_ITEMS are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

Modifying a Program Definition

After a program is defined, you can change the definition with the Modify Program statement. When modifying a program that is used to launch an application, however, consider upward and downward compatibility between software versions.

The following statement lets you add or remove defining clauses and change the value of clause arguments:

```
modify program NAME [MOD_ITEM] {MOD_ITEM};
```

NAME is the name of the program you want to modify.

MOD_ITEM is the type of modification you want to make.

You can make the following modifications. Each is specified in a Modify Program clause, as listed in the following table. Note that you only need to specify fields to be modified.

Modify Program Clause	Specifies that...
code CODE	The current code definition changes to that of the new code entered.
description VALUE	The current description, if any, changes to the value entered.
external mql java	The specification of the program type (external or MQL or Java) changes as entered.
file FILENAME	The contents of file are added to the code section of the program.
icon FILENAME	The image is changed to the new image in the file specified.
add rule NAME	The named rule is added.
remove rule NAME	The named rule is removed.

Modify Program Clause	Specifies that...
name NAME	<p>The current name of the program changes to the name entered.</p> <p>Note: If you rename a program, it may become available within certain eMatrix features. For example, if you rename a program that is part of a toolset, the program will need to be added to the toolset again.</p>
[!]needsbusinessobject	<p>The status of the need for a business object changes as indicated here:</p> <p>needsbusinessobject is used when a business object is needed.</p> <p>!needsbusinessobject (or notneedsbusinessobject) is used when a business object is not needed.</p>
[! not]downloadable	<p>The status of downloadable changes as indicated here:</p> <p>downloadable is specified when the program includes code for operations not supported on the Web product (for example, Tk dialogs or reads/writes to a local file).</p> <p>!downloadable (or notdownloadable) is specified when the program does not include code for operations not supported on the Web product.</p>
[! not]pipe	<p>The external program uses the "piped" service or not.</p>
[! not]pooled	<p>The program uses pool interpreters or not.</p>
execute immediate	<p>The status of program execution changes so the program runs within the current transaction.</p>
execute deferred	<p>The status of program execution changes so the program runs only after the outermost transaction is successfully committed.</p>
hidden	<p>The hidden option is changed to specify that the object is hidden.</p>
nohidden	<p>The hidden option is changed to specify that the object is not hidden.</p>
property NAME [to ADMINTYPE NAME] [value STRING]	<p>The named property is modified.</p>
add property NAME [to ADMINTYPE NAME] [value STRING]	<p>The named property is added.</p>
remove property NAME [to ADMINTYPE NAME] [value STRING]	<p>The named property is removed.</p>

Each modification clause is related to the arguments that define the program.

Using an External Text Editor for Making Tcl Program Changes

The following is a simple methodology that enables you to use the text editor of your choice while making changes to eMatrix Tcl programs:

1. Create the program "code" in a file that is named the same as the program object in business.
2. Create a separate Tcl program (for example, UpdatePrograms.tcl) that lists all the programs, each within an "mql modify program...file..." like what follows:

```
puts stdout ">>Modifying ProgramName.tcl"
set mqlret [ catch { eval mql modify program 'ProgramName.tcl' \
file \"e:/meyerj/matrix-dev/solutions/src/ProdCfg/ProgramName.tcl\" \
} outStr ]
```

3. You can keep a separate MQL session open where you can run the "UpdatePrograms.tcl" anytime, to test and see the effects of your changes. This program then updates all the programs that are listed with the latest code that is stored on disk. It generally takes about 5 to 30 seconds, depending on the number of programs listed.

This method also works well when you need to use a real source control mechanism because the program code and the updating program file are stored on the disk.

Note: If you are using the above methodology, you should never update program code directly in Business Modeler, since the source for the program is stored on your personal computer rather than in the database.

Deleting a Program

If a program is no longer required, you may delete it with the Delete Program statement:

```
delete program NAME;
```

NAME is the name of the program to be deleted.

When this statement is processed, eMatrix searches the list of programs. If the name is not found, an error message is displayed. If the name is found, the program is deleted.

20

Working With Event Triggers

Overview of Event Triggers	516
Trigger Scenarios	517
Transaction Boundaries and Events.....	518
Transactions Boundaries and Action Triggers.....	518
Designing Triggers	520
Supported Events	520
Types of Triggers.....	521
Multi-Trigger Events.....	522
User Interaction	524
Lifecycle Checks and Actions with Event Triggers	525
Trigger Programs	526
Trigger Macros.....	526
Check Programs.....	526
Override Programs	527
Action Programs	528
Validate Query Trigger	529
Enabling the Validate Query Trigger.....	529
Using the ValidateQuery Trigger with matrixcgi.....	530
Writing the ValidateQuery Program.....	530
Use Case	531
Recursion Detection Modes and Limits	535
Limiting the Trigger Stack Depth	536
Override Example.....	536
Logging Trigger Events.	539
Enable/Disable Triggers	540

Overview of Event Triggers

Event Triggers can be used to implement business rules. The eMatrix concept of Checks and Actions has been extended in Triggers to include a much larger class of events. Where Check and Action Programs are executed upon an object's promotion to another State, Trigger Programs can be executed by other lifecycle events such as rejection, scheduling, and approval. Further, most of the events associated with Business and Relationship objects, such as creation, deletion, and modification of attributes, also support Triggers.

Event Triggers provide a powerful way to customize eMatrix behavior through Program objects. Triggers may contain up to three Programs which can all work together, or each work alone, to blend functionality and control in eMatrix. Programs can be written that modify the behavior of many functions, and that behavior can vary depending on the circumstances. As such, programmers are actually creating *hybrid* versions of eMatrix when they implement Triggers and should therefore be the source for troubleshooting any problems that may arise.

Because of their complexity, Event Triggers should be thoroughly tested in a development environment prior to implementation in a production database.

Trigger Scenarios

With the power of Event Triggers, it is possible to programmatically guarantee certain events cause (or are replaced by) other actions. Some scenarios for triggers are listed below.

- An Attribute range could be updated upon creation of an object. For example, when a Vendor object is created, its name is added to the “Provider” Attribute range.
- When an Attribute is modified, a connection could occur. For example, when the Provider Attribute field of a Purchase Order object is entered, the P.O. could be connected to the Vendor object.
- Relationships could be given a lifecycle. For example, three relationships could exist called “Proposed”, “Planned” and “Complete.” A Project object could be connected to various Tasks with the Proposed relationship. When the Task has been worked out, the two objects are connected with the Planned relationship, which triggers a disconnect of the Proposed relationship. Likewise, when the Complete connection is created, the Planned connection is removed automatically.
- The changename event could trigger the name of a related object to be changed. For example, if a Product name is changed, the Manual object which will document it could be automatically changed, and notification sent.
- The checkout event could trigger a program that would check to see if the object is locked and prompt the user to lock if they want to check the file back into the object.
- The checkin event could trigger the object to be unlocked.
- When objects are deleted, if files exist, they can programmatically be copied to a specified location or externally be emailed to the System Administrator.
- When one state is scheduled, each successive state could be scheduled automatically by a specified offset value.
- Objects could be added or removed from sets when an event occurs. For example, the set called “Pending Projects” could be modified as objects reach the “Complete” state.

Transaction Boundaries and Events

When designing Triggers, it is important to understand the eMatrix transaction model. The discussion below is supplemental to the description in *Working With Transactions* in Chapter 2.

Over time, an object may transition from one state to another, as its properties are changed by the occurrence of events. The use of the term *state* here is more general than the States in an eMatrix Policy. For example, changing the description of a business object is an event that changes the state of that business object, but the lifecycle State probably remains the same.

In eMatrix, each event is implicitly wrapped in a transaction boundary and the achievement of a new state is not complete until the transaction is committed (thus ensuring database consistency). Common eMatrix functions are often actually a series of actions, which rely on the proper execution of all of the previous actions. For example, checking access and logging history are actions that are implicitly wrapped in the transaction boundary of most eMatrix operations. The design guarantees that objects are not updated without the history being logged or vice versa.

However, eMatrix also allows transaction boundaries to be explicitly defined, so that programmers can roll back all related actions if one of them failed. When transaction boundaries are implicit, the state of an object is considered changed as soon as an event transaction takes place. But, in order for an explicitly-defined transaction to be effective, each of its commands must be successful.

Transactions Boundaries and Action Triggers

As you will see in later sections, when we add Event Triggers to the eMatrix transaction model, the internal logic looks generally like this:

- 1) Event transaction is started
- 2) Normal processing of access checking occurs.*
- 3) If it exists, the Check Trigger is fired.
- 4) If Check blocks, then transaction aborts.
 If not, the Override Trigger is fired if it exists.**
- 5) The Event transaction is then committed regardless of
 an override or normal activity.
- 6) Finally, if there is an Action Trigger, it is fired.

*Access is checked but the event is not stopped if no access. The RPE CHECKACCESSFLAG and ACCESSFLAG are set so the check and override events can determine current state and correct action.

**At this point, the lack of access will stop the event if the override doesn't override the event with some other operation.

If an Event is part of a larger transaction, Action Programs may be defined with the “Execute Deferred” flag so that execution is deferred until after all transactions are committed. So, the transaction model now looks as follows:

- 1) Larger transaction is started with any number of commands before the Event with the trigger.
- 2) Event transaction is started.
- 3) Normal processing of access checking occurs.
- 4) If it exists, the Check Trigger is fired.
- 5) If Check blocks, then transaction aborts.
If not, the Override Trigger is fired if it exists.
- 6) The Event transaction is then committed regardless of an override or normal activity.
- 7) If the Event Trigger has a non-deferred Action Program, it is executed.
- 8) The larger transaction is committed.
- 9) Finally, if there is a deferred Action Program, it is now triggered.

Deferred Action programs will queue up if more than one is deferred within a transaction. This ensures that the order of execution remains the same as the Events which triggered them.

Designing Triggers

Supported Events

Most of the events associated with Business Objects, Relationship Objects, Workflows, and Policy lifecycle events support Triggers. Triggers are added to Type, Relationship, Attribute, Process, and State definitions in much the same way as Attributes and Methods are added to the Type definition, and when used in Types, they are likewise inherited.

Business Object Events

The following Business Object events support Triggers. These Triggers can be added to Type definitions.

changename	changeowner	changepolicy
changetype	changevault	checkin
checkout	connect	copy
create	delete	disconnect
grant	lock	modifyattribute
modifydescription	removefile	revision
n		
revoke	unlock	

Refer to [Working With Types](#) in Chapter 13 for more information and syntax for adding triggers to Type definitions.

Attribute Events

Attributes themselves can have triggers on the modify event.

*Note that triggers can be defined on business object or relationship modifyattribute events, which are fired whenever **any** attribute is modified (once per attribute modified), or on an **attribute's** modify event, which is fired when that attribute in particular is modified.*

Relationship Events

The following relationship events support Triggers, and can be added to Relationship definitions.

freeze	create (connect)	modifyattribute
thaw	delete (disconnect)	

Refer to [Overview of Relationships](#) in Chapter 14, for more information and syntax for adding triggers to Relationship definitions.

State Events

The following lifecycle events support Triggers. These Triggers can be added to Policy definitions.

approve	demote	disable
enable	ignore	override
promote	reject	schedule
unsign		

Refer to [Working With Policies](#) in Chapter 17, for more information and syntax for adding triggers to Policy definitions.

Workflow Events

The following Workflow and Activity events support triggers. These Triggers are defined in the Process and Activity objects.

Process Events

FinishProcess	ReassignProcess	ResumeProcess
StartProcess	StopProcess	SuspendProcess

Auto Activity Events

ActivateAutoActivit Y	CompleteAutoActivit Y	OverrideAutoActivit Y
ReassignAutoActivit Y	ResumeAutoActivity	SuspendAutoActivity

Interactive Activity Events

ActivateActivity	CompleteActivity	OverrideActivity
ReassignActivity	ResumeActivity	SuspendActivity

Refer to [Overview of Workflows](#) in Chapter 33 for more information on adding triggers to processes and activities.

Types of Triggers

Each supported data event has three types of triggers:

- a **Check** trigger that fires before the event occurs
- an **Override** trigger that can replace the event transaction
- an **Action** trigger that fires after the event occurs

For each event, Programs can be specified for none, any, or all of the Trigger types.

Check Trigger

The check trigger executes first, before any of the normal system code which carries out the event has a chance to execute.

The program specified here can be used in two ways:

- This program can do pre-event processing in anticipation of the event.
- It can also block the event from actually occurring, aborting the event transaction, which means that no other trigger programs are get executed.

Refer to [Check Programs](#) for more information.

Override Trigger

The override trigger is fired after the check trigger, but before any of the normal system code that carries out the event has a chance to execute. It can also be used in more than one way:

- It can perform pre-event processing after the check program has decided not to block the event.
- It can replace the normal event processing, with an alternate event.
- It can also be used to perform a “soft block” on the event. This means that the event is blocked without raising an exception, and so the action program is executed and any pre-processing done in the check program is not rolled back. Refer to [Blocking Events](#) for more information.

Refer to [Override Programs](#) for more information.

Action Trigger

The action trigger fires after the event transaction is committed, whether the normal event took place or the override program replaced the event.

The action program typically extends the functionality associated with an event (or the override program that replaced the event). A program attached to the action trigger can perform any post-event processing that must be done after the event has updated the database. For example, an Action Program may generate a report in an external application that lists the changes that have just occurred. Since you would not want this action to fire unless those changes actually took place, the Action Trigger is always fired after the event transaction is committed.

Consider, however, that the same Event is part of a larger transaction. If the Action Trigger fired immediately after the Event, generating a report outside of eMatrix, what would happen if the transaction failed? eMatrix can only roll back internal state changes, so the report would be generated erroneously. In this case, the Action Program execution should be defined as “Deferred” so that it is deferred until all transactions are committed.

Refer to [Action Programs](#) for more information.

Multi-Trigger Events

When designing Triggers, programmers should keep in mind that several Triggers may be fired by what appears to be a single event. For example the “Open for Edit” Event appears to the user to be one operation, but under the covers, the system performs four Events (checkout, lock, checkin, and unlock), each of which may have three Triggers fire (check,

override, action). Trigger programmers should carefully evaluate exactly where (which event and trigger) their logic should be introduced. Also, remember that if programs exist in many of these Triggers, performance will be affected.

Two Phases of Create Event

The creation of a business object can be accomplished in either MQL (with the `add businessobject` statement) or within eMatrix (using the Object/New menu selection). In both cases, it is a two-phased process: first, the creation phase, and then the “modification” phase. Likewise, creating clones or revisions (`copy businessobject` or `revise businessobject`) as well as connections (`connect businessobject`) are also accomplished in two phases.

When designing triggers, it is important to understand what occurs when, since a non-deferred program may require data that has not yet been entered into the database. For purposes of this discussion we will use the `businessobject` creation event, but keep in mind that the same applies to connections as well.

During the creation phase, the object is given a type, name, revision, policy, and vault. At this point, any non-deferred action program assigned to the initial state of the object is executed, followed by the execution of the non-deferred create action program, if any. Since the object has not yet been given anything more than an identity, only very basic macro values (see the [Macros Appendix](#)) have any meaning at this point, and so these action programs must rely only on the information available.

In the modification phase, the description, attribute values, owner, and schedule information (if any, from MQL) are defined. During this phase many events may occur. There will be a `modifyattribute` business object event for each attribute (as well as a `modify` event on the attribute) having its value set. There may also be a `changeowner` event, `modifydescription` event, and any number of schedule events. Thus, any check, override, and non-deferred action programs associated with these types of triggers will execute during this modification phase. Because of this, action programs which at first seem appropriately placed on the create event, may actually work well on another event that occurs during the modification phase.

Another aspect of the create event has to do with transaction boundaries. Transaction boundaries for creating business objects differ depending on which eMatrix application is used (eMatrix or MQL). The create and modification events are one transaction in MQL, and split into two in eMatrix. In either case, if the creation phase fails the business object is not created. However, if the modification phase fails, in eMatrix the object will still exist; in MQL it won’t. If the execution of any action programs is deferred until after the outer transaction commits, they will behave differently when using MQL and the eMatrix GUI:

- In the eMatrix GUI, any deferred state action program and/or create action trigger will execute before the modification phase.
- In MQL, they will execute after the modification phase.

In summary, when writing programs for use in the create and modify business object events, keep in mind the following:

- The creation of an object is done in two phases: the creation of the object, then the modification of the object’s attributes. During the creation phase, the following occurs:
 1. First, the create check trigger program executes.
 2. Next, the create override trigger program executes.

- 3. The create event then occurs.
- 4. Any action defined on the first state in the lifecycle then executes.
- 5. Finally, the non-deferred create action trigger program executes.
- Programs written for use as create event action triggers must use only those macros that pass very basic information since the object knows only its type, name, revision, policy, and vault. Programs that require more information should probably be placed as triggers on another event occurring during the modification phase.
- The create and modification events are one transaction in MQL, and split into two in eMatrix, and therefore deferred action programs will be executed after different phases depending on which application is used.

Relationship Triggers

Relationship Events pose an interesting dilemma when used with Triggers. When forming or breaking a relationship between two business objects, Trigger programs are fired in the following order:

1. The connect or disconnect Check of the FROM Business Object
2. The connect or disconnect Check of the TO Business Object
3. The create or delete Check of the Relationship
4. The create or delete Override of the Relationship
5. The connect or disconnect Action of the FROM Business Object
6. The connect or disconnect Action of the TO Business Object
7. The create or delete Action of the Relationship.

Notice that the Override Triggers on the FROM and TO Objects are NOT fired.

Override Programs specified in Business Object Connect/Disconnect Events are NEVER fired. The Override Program required for establishing or breaking relationships should be specified in the Relationship create/delete Trigger.

Consider the consequences of the connection operation being replaced at one end (by an Override program) but not at the other. The resulting Relationship would be invalid, since it would be missing an end. For this reason, and to avoid unnecessary complexity, only one Override program is available for create/delete connection events. And, since Relationship Trigger programs have access to macros which can provide information about both connection ends, one Override is enough. With three chances to block, a way to replace, and three separate actions available, there should be enough hooks to satisfy most Trigger programmers.

User Interaction

Triggers should be used to perform “behind the scenes” operations on data, and should not require any interaction with the end user. Since triggers already enlarge the event transaction, you would not want to tie up database resources any longer than necessary. For example, creating an object locks a row in the Oracle table corresponding to the object type. In order to maintain database integrity, other sessions that are attempting to access that table must wait until the create transaction is committed. If a trigger program creates a business object, it should avoid presenting any user interface (for example, a Tk dialog

box, or a wizard) since this results in locking the Oracle table until the user dismisses the dialog.

If it is *absolutely* necessary for a trigger to interact with the user, this condition can be avoided by deferring the action trigger to run outside the current transaction, by using the “Execute Deferred” setting in the program definition.

Refer to [Transaction Management](#) and [Execute Clause](#) in Chapter 19.

Lifecycle Checks and Actions with Event Triggers

Existing Policies may employ Checks and Actions as defined before Event Triggers were available. These Programs are still valid and are executed in conjunction with Event Triggers as explained below. It is advisable, however, that new Checks and Actions are attached as Triggers, since greater functionality is available there.

- 1) Promote event transaction is started
- 2) State1 promote Check Trigger is fired, if it exists.
- 3) If Check blocks, then transaction aborts.
If not, State1 promote Override Trigger is fired if it exists.
- 4) If event is not replaced:
 - 4.1) Promote Access is checked --> abort if no access
 - 4.2) State transition requirements satisfaction checked --> abort if not satisfied.
 - 4.3) State transition Check program executed (OLD CHECK) --> abort if fails.
 - 4.4) Is State1 disabled? --> abort if yes.
 - 4.5) Promote object to State2
 - 4.6) State2 Notification and Routing occurs, if any.
 - 4.7) State2 Action program executed (OLD ACTION)
 - 4.8) Add promote history to object.
- 5) Promote transaction committed.
- 6) State1 promote Action Trigger fires.

Note that State1’s Action Trigger fires even though the “old” lifecycle Action of State2 has already executed. Note also that lifecycle Actions are part of the event transaction. This means that a failure in the lifecycle Action program would result in the promote transaction being rolled back. Also, the exit code of a lifecycle Action is checked, and if it returns a non-zero integer, the promotion is also rolled back. Refer to [Program Exit vs. Program Failure](#) in Chapter 19 for more information.

Trigger Programs

A few points need to be made about Program objects that are attached to Event Triggers:

- Any changes made to the Program object are immediately picked up when fired by a trigger. This implies that a check or override program could actually modify the code of an action program that follows, which allows truly dynamic behavior to be designed.
- Check and Override programs must return an exit code in order for the event processing to know how to proceed. However, code can be incorporated into the Program that performs other functions prior to setting the exit code's integer value. Refer to [Program Exit vs. Program Failure](#) in Chapter 19 for more information on returning exit codes.
- Programs that use Trigger-specific macros cannot be used as methods, since the macro values rely on the triggering event.
- When a Program object that is attached to a trigger is renamed, the newly named Program object continues to be attached to the trigger.
- Deleting a Program object that is attached to a trigger automatically detaches the Program object from the trigger.

Trigger Macros

Macros are a simple name/value string substitution mechanism, where the macro value is substituted for the macro name when used in a Program as follows:

```
$ {MACRONAME}
```

This single one-pass string substitution process occurs just prior to program execution. Refer to the [Macros Appendix](#) for more information.

Check Programs

Trigger Check program execution should never be deferred. The exit code should be an integer value, which is always tested by the trigger logic to determine whether or not to block the event. Any system errors encountered while executing the check program will abort the event transaction.

Blocking Events

When the check program returns a non-zero exit code, the event is not performed. This is one way to block events from occurring. Unlike replacing events (see below), blocking events with a Check program causes the event transaction to abort, which raises an exception (and therefore displays an error). If a check program blocks, then neither the override program nor the action program get executed.

Events can also be blocked using an Override Program. To block an event without having an error message presented, use an override trigger. If the exit code from the override program is a non-zero integer, the event is completely replaced, but the transaction is committed, and no error message is displayed.

In this scenario the normal event processing is skipped, and therefore the override program *becomes* the event. However, if the override program causes the event to be

replaced without providing code for an alternative event, the original event is, in a sense, blocked. This is called a “soft block.” It is different from blocking with the Check program in that the event transaction is committed anyway. Since the transaction is not aborted, any processing that may have occurred in the Check program is not rolled back and the action trigger will fire. If there is an action trigger program, you may need to use the RPE to communicate to the action program that the event was blocked, and not to do its normal activity.

Refer to [Program Exit vs. Program Failure](#) in Chapter 19 for more information on returning exit codes.

Override Programs

Override program execution should never be deferred. The exit code should be an integer value, and is always tested by the trigger logic to determine whether to replace the event or not. Refer to [Generating an Exit Code](#) in Chapter 19 for information on returning an exit code.

If the override program returns a zero exit code, the event is not replaced, and the flow of control proceeds in its normal fashion (that is, the normal system code is executed, the history of the event is recorded, the transaction is committed, and then the Action trigger is fired, if it exists.). This would be how one adds pre-event processing to an eMatrix event. Refer to [Program Exit vs. Program Failure](#) in Chapter 19 for more information on returning exit codes.

Replacing Events

When the override program returns a non-zero exit code, the normal system event code and history mechanism is skipped, and then the event transaction is committed and the action program is executed. This is how one can replace existing event handling with new, custom event handling. A non-zero exit code in an override program can also be used to block the event. Refer to [Blocking Events](#) above.

When replacing normal event processing, the associated history entry is not recorded. This brings up the issue of which events should occur in the override program. Certainly if the overriding behavior causes other *normal* eMatrix events to occur, those events will be logged by their associated history mechanism. If this is not the case, the overriding behavior will go unrecorded as far as the history mechanism is concerned. Of course, other means of recording the event can be done by the check or action code. The following MQL command can be used to add custom history entries to a business object:

```
modify bus TYPE NAME REV add history VALUE [comment VALUE];
```

For details, see [Working With History](#) in Chapter 36.

Changing the behavior of eMatrix without any explicit indication, as is the case when using the replace event feature, can potentially lead to problems that are extremely difficult to troubleshoot. Therefore, when triggers are implemented that are designed to replace expected eMatrix behavior, it is recommended that a trigger log is created by setting the environment variable MX_TRIGGER_LOG to TRUE.

Any system errors encountered while executing the override program will abort the event transaction.

Action Programs

Without transactions and potential rollbacks to worry about, it would clearly be best to execute the action program immediately after the event takes place. This is the case with non-deferred action programs. Refer to [Action Trigger](#) for more information.

Trigger Action program execution is deferred if the “Execute Deferred” setting is on. Deferred action programs are queued for execution which takes place when the outer-most transaction is committed. See [Execute Clause](#) in Chapter 19 for additional information.

Trigger Action Program exit codes are never tested by the trigger logic. (This is not the case, however for lifecycle Actions. In a state promotion action, if the program returns 1 or aborts, the promote transaction is aborted.)

Any system errors encountered while executing a non-deferred action program will only abort the event transaction if it is nested within an outer transaction.

Validate Query Trigger

You can add a trigger to the query execution event so that administrators may control the types of queries that users are allowed to perform. Trigger programmers can customize the behavior of queries (even block them from evaluating) depending on the query patterns specified. The trigger is fired from the Find or Find Like windows in both the desktop and Web versions of eMatrix, where wide-open query definitions are allowed, as well as on specific queries in MQL or ADK programs. The Validate Query trigger has two main purposes:

- To block certain types of queries that perhaps are inefficient
- To change the query patterns in the background to perform a more specific query than originally attempted. For example, if the vault pattern is a wildcard (*), the value can be changed to be a list of all local vaults instead.

In addition, customized error and notice messages can be created. For example, in the case of a blocked query, the program may display a message that indicates the reason for blocking it, such as more information is required.

The validate query trigger is different from other eMatrix triggers. It executes a single program that works as a combined check and override trigger, and is designed as a way to modify the current query (via the RPE).

Enabling the Validate Query Trigger

The validate query trigger is enabled in the desktop application by default, as long as a program object named ValidateQuery exists in the database. (Refer to [Writing the ValidateQuery Program](#)). Because an additional call is made to the database that may affect performance, particularly through a server, query triggers must be enabled for use in the Web version of eMatrix, by adding the following to the HTML page that starts the eMatrix:

```
document.writeln("<param name='QueryTrigger' value='True'>");
```

If you will use custom applets, your pages may use the straight HTML syntax:

```
<PARAM name="QueryTrigger" value="True">
```

In MQL, the evaluate query and temp query commands can include the querytrigger clause, which will cause the ValidateQuery program to be executed, provided it exists. For example:

```
<mql> temp query bus * * * querytrigger vault v1;
<mql> evaluate query q1 querytrigger;
```

Even with triggers turned off, when querytrigger is included in a query command, the ValidateQuery program gets run.

Program objects that perform queries are not affected by default in either the desktop or Web version of eMatrix. If required, the querytrigger flag must be included in the program code.

Using the validate query trigger in ADK programs

Two ADK methods are available to set and get the query trigger flag. By default the flag is set to false. The methods are:

```
public setQueryTrigger(boolean queryTrigger)
public boolean getQueryTrigger ()
```

The following example performs a temporary query and the validate query trigger will fire as long as the program object exists:

```
Query myQuery = new Query("");
myQuery.setType("*");
myQuery.setName("*");
myQuery.setRevision("*");
myQuery.setVault("*");
myQuery.setQueryTrigger(true);
myQuery.evaluate(getContext()));
...
...
```

Using the ValidateQuery Trigger with matrixcgi

http://appserver:port/cgi-bin/matrixcgi.exe?appl=icons&type=T*&querytrigger=true

If not included, the querytrigger parameter is set to false. Note that the querytrigger parameter is only valid when the appl=icons parameter is also included.

Writing the ValidateQuery Program

The validate query trigger is fired before the query is evaluated. The program will generally be designed to check the query patterns specified. This may be accomplished because of the availability of [Query Event Macros](#).

If the ValidateQuery program is a JPO, no macros are available as input arguments. To work around this issue, the ValidateQuery program should be a Tcl program that calls the JPO to do the actual work.

When changing query patterns, you must use global variables. For example, to change the vault pattern in the background if only a wildcard is provided, you could use:

```
tcl;
eval {
    set sVault [mql get env VAULT_PATTERN ]

    if { $sVault=="*" } {
        set sVault "vault1,vault2,vault3"
    }

    mql set env global VAULT_PATTERN

    exit 0
}
```

As is the case with other event triggers, the exit code determines if the event is blocked or overridden. If the exit code is a zero, the query is performed; the global query pattern RPE variables are read and used by the query, which allows the program code to change them before the query is evaluated. (Remember, the validate query trigger is really a

combination of a check and an override trigger.) If the exit code is a non-zero value, a soft block occurs. In this case the query as entered is not executed; the query event is overridden and the transaction is committed, closing the Find or Find Like window. It is up to the trigger program code that overrides the query to generate a message to inform the user that the query did not get executed.

While overriding the query code with other events is possible, the main purpose of a query trigger is to block unqualified queries and modify the query patterns via RPE variables. If overriding the query code, query performance, Tcl limitations (such as the fact that Tcl is single threaded) and recursive queries must be considered. The query is a fundamental part of eMatrix and overwriting the code must be done carefully. To return business objects to the current eMatrix Window in eMatrix desktop or eMatrix applet, the MQL commands, `appl icons` and `appl details` can be used to populate the window with business objects. Refer to *Dialog Scripting Language* in Chapter 19 for more information.

Use Case

The validate query trigger can be used to ensure that user-defined queries satisfy some pre-defined requirements. For example, the program below is one solution for ensuring that:

- All queries in local vaults have at least one filter (i.e., type, name, rev, or owner for basic queries), but partial wildcards are permitted.
- A query on all vaults (*) does not include foreign vaults.
- Queries on a specified foreign vault have both type and name filters.

The ValidateQuery program to satisfy above:

```
tcl;

eval {
    eval [ mql print program utList.tcl select code dump ]

#####
#proc tclListToMxList {tclList} {

    set mxList ""

    foreach i $tclList {
        append mxList $i
        append mxList ","
    }
    set length [expr [string length $mxList] - 2 ]
    set mxList [ string range $mxList 0 $length ]

    return $mxList
}

#####
#proc mxListToTclList {mxList} {

    set tclList [ split $mxList "," ]

    return $tclList
}

#####
```

```

proc MatchingVaults {sPattern tclVaultList} {
    set lVaultsToBeSearched ""

    set firstIndex [ lsearch $tclVaultList $sPattern ]

    while { $firstIndex != -1 } {
        lappend lVaultsToBeSearched [ lindex $tclVaultList $firstIndex]

        set index [expr $firstIndex + 1 ]

        set tclVaultList [lrange $tclVaultList $index end]
        set firstIndex [ lsearch $tclVaultList $sPattern ]
    }
    return $lVaultsToBeSearched
}

#####
proc onlyWildcardsInString {sString} {

    set wildcard "*"
    set bReturn FALSE

    set iFirstIndex [string first $wildcard $sString]

    if {$iFirstIndex != -1 } {

        # To discover many stars (ex ***)
        set bReturn TRUE

        for {set i 0} {$i < [string length $sString]} {incr i} {
            if { [string index $sString $i] != $wildcard } {
                set bReturn FALSE
                break
            }
        }
        return $bReturn
    }

    #####
    #
    # Variable definition
    #
    #####
}

# Default values
set iExit 0
set sErrorMessage ""

# Get all vaults from Matrix
set sAllVaultsList [ mql list vault ]
set lAllVaultsList [ split $sAllVaultsList \n ]

# Defined adaplets
set lAdapletVaults {a1 a2}
set lRemoteVaults {r1 r2}

# Local vaults

```

```

set lLocalVaults [ utListSubtract $lAllVaultsList [ concat $lAdapletVaults
$lRemoteVaults ] ]

# Get query env
set sVault [mql get env VAULT_PATTERN ]
set sType [mql get env TYPE_PATTERN ]
set sName [mql get env NAME_PATTERN ]
set sRevision [mql get env REVISION_PATTERN ]
set sOwner [mql get env OWNER_PATTERN ]

set lVaultsToBeSearched {}

#####
#
# Find out which vault(s)are going to be queried
#
#####

# If vault pattern is a wildcard change the vault to only search local vaults
if { [onlyWildcardsInString $sVault] } {
    set sVault [ tclListToMxList $lLocalVaults ]
}
mql set env global VAULT_PATTERN $sVault

# if vault pattern is a comma separated list of vaults
if { [string match *,* $sVault] == 1} {
    set lVaultList [ split $sVault "," ]
    foreach v $lVaultList {
        set lVaultsToBeSearched [ concat $lVaultsToBeSearched [MatchingVaults $v
$lAllVaultsList ] ]
    }
}

# not a list of vaults
} else {
    set lVaultsToBeSearched [MatchingVaults $sVault $lAllVaultsList ]
}

#####
#
# Go through the vaults that are going to be searched and
# check if all criteria are satisfied
#
#####

set bGeneralCheck FALSE

foreach v $lVaultsToBeSearched {

    # Foreign vault a1 is not allowed to have * only ? is allowed
    if { $v == "a1" } {
        if { ([string first * $sType] != -1) && ([string first * $sName] != -1)} {
            set iExit 1
            append sErrorMessage "Query not allowed. When searching in Vault $v, none
of the query patterns type and name is allowed include a wildcard (*). Use one or many
question marks (?) for unknown characters instead\n\n"
        }
    } elseif {$v == "a2"} {
        if { [string first * $sName] != -1 } {

```

```

        set iExit 1
        append sErrorMessage "Query not allowed. When searching in Vault $v, the
query patterns type is allowed include a wildcard (*). Use one or many question marks
(?) for unknown characters instead\n\n"
    }
} else {
    set bGeneralCheck TRUE
}
}

if {$bGeneralCheck} {

    set iAllowedWildards 3
    set iCurrent 0

    if { [onlyWildcardsInString $sType] } {
        incr iCurrent
    }
    if { [onlyWildcardsInString $sName] } {
        incr iCurrent
    }
    if { [onlyWildcardsInString $sRevision] } {
        incr iCurrent
    }
    if { [onlyWildcardsInString $sOwner] } {
        incr iCurrent
    }
    if {$iCurrent > $iAllowedWildards } {
        set iExit 1
        append sErrorMessage "Query not allowed. One of the query patterns type, name,
revision or owner must be more qualified than a wildcard (*)\n\n"
    }
}

#
# Pop up error message if needed
#
if {$iExit == 1} {
    mql error $sErrorMessage
}

exit $iExit
}

```

Recursion Detection Modes and Limits

eMatrix allows Program objects to execute themselves. This is what we will call *direct*, or *explicit recursion*. Explicit recursion is easy to spot and is a powerful mechanism when used properly. On the other hand, Event Triggers introduce what we will call *indirect*, or *implicit recursion*. Implicit recursion is caused when a trigger program generates the same event with which it is associated. This form of recursion is more difficult to spot, and may or may not be useful. What can make implicit recursion difficult to spot is that the event that triggers the running program may be part of a program that was triggered by a totally different event. The amount of indirection could be many levels deep, and nearly impossible to figure out.

You can configure how your system detects trigger recursion by setting the MX_TRIGGER_RECURRENCE_DETECTION environment variable. There are 3 scenarios:

1. By default, this variable is set to **name**, and all recursion encountered through Event Triggers is treated as a no-op (that is, if an attempt is made to execute a running trigger program, the attempt fails but is not treated as an error). The system stores the name of the executing program, and uses a name matching test to determine if recursion is being attempted. If a case of recursion is encountered, the system simply skips execution of that trigger program but otherwise continues to process in non-error mode.
2. Since programs can be configured with input arguments, it is reasonable to design programs for reuse, and therefore, it may be desirable to allow running programs to be executed again. For example, you may have a program that promotes an object that it is passed. When this program is executed, it could fire a promote trigger that calls the same program, but passes in a different object. To allow this, you could set eMatrix to use a signature matching test by setting the variable to **signature**. When signature matching is enabled, the system stores the following five properties of a trigger program when it is executed:
 - Program Name
 - Input Arguments
 - Trigger Event
 - Trigger Type
 - Target object IDIf a program attempts to execute where all five of these properties match another program on the running stack, recursion is detected and the routine is skipped.

The target object ID is a unique internal number that identifies the business object or relationship instance that is the target of the given event. When the event is “create” on a business object or on a relationship instance, a unique number is generated but this number does not identify an object (since one does not yet exist). This means that there is a much higher potential for unwanted recursion in these two cases so special attention should be paid when writing trigger programs for these two cases.

3. When full recursion is desired, the detection of recursion can be turned off by setting the variable to **none**. In this case it is very important to avoid infinite recursion using the environment variable MX_TRIGGER_RECURRENCE_LIMIT. See below.

Limiting the Trigger Stack Depth

The MX_TRIGGER_RECURSION_LIMIT variable sets the number of trigger programs that may be placed on the runtime stack to the value set, helping to avoid server crashes due to the host system running out of stack space. The default value is 30 (which may be too high, particularly when using the Web version of eMatrix through a Windows server). When the limit is reached, a warning is added to the trigger trace file such as:

```
"WARNING: Override Trigger for <event>: <program_name> has exceeded recursion limit!"  
"WARNING: Validate Query for <event>: <program_name> has exceeded recursion limit!"
```

The limit is used in conjunction with MX_TRIGGER_RECURSION_DETECTION, no matter what type of recursion detection is enabled.

Override Example

Now let's take a look at what happens when Trigger Programs issue commands that cause events to occur that also have Triggers. Our example involves the business object "Document Checker 0" and a bit of a squabble between three users, "Tom", "Dick", and "Harry," who fight over ownership. The default recursion test (name matching) is in use.

Assume that the following Triggers are in place in the Document type on the changeowner event:

```
# Changeowner Event Check Program on Document Type:  
mql  
code "output 'DocumentChangeOwnerCheck: OBJECT = ${OBJECT} TYPE = ${TYPE}';  
output ' NAME = ${NAME} REVISION = ${REVISION} OWNER = ${OWNER}';  
output ' ACCESSFLAG = ${ACCESSFLAG} USER = ${USER}';  
output ' CHECKACCESSFLAG = ${CHECKACCESSFLAG}';  
output ' TIMESTAMP = ${TIMESTAMP}';  
output ' NEWOWNER = ${NEWOWNER}';  
tcl;  
set status 0  
set status $env(RETURN_CHECK_CODE)  
exit $status  
";  
#Changeowner event Override Program on Document #Type:  
mql  
code "output 'DocumentChangeOwnerOverride: OBJECT = ${OBJECT} TYPE = ${TYPE}';  
output ' NAME = ${NAME} REVISION = ${REVISION} OWNER = ${OWNER}';  
output ' ACCESSFLAG = ${ACCESSFLAG} USER = ${USER}';  
output ' CHECKACCESSFLAG = ${CHECKACCESSFLAG}';  
output ' TIMESTAMP = ${TIMESTAMP}';  
output ' NEWOWNER = ${NEWOWNER}';  
modify bus ${TYPE} ${NAME} ${REVISION} owner Dick;  
tcl;  
set status 0  
set status $env(RETURN_OVERRIDE_CODE)  
exit $status  
";
```

The exit code from the check and override Trigger programs is handled by environment variables. Their settings are: RETURN_CHECK_CODE=0 and RETURN_OVERRIDE_CODE=1 (which means the check program will not block and the override program will replace the normal event handling). Harry is the current owner of the Document object and has decided to change ownership to Tom. The output is shown below.

```

MQL<7>modify bus Document Checker 0 owner Tom;
DocumentChangeOwnerCheck: OBJECT = Document Checker 0 TYPE = Document
  NAME = Checker REVISION = 0 OWNER = Harry
ACCESSFLAG = True USER = creator
  CHECKACCESSFLAG = True
  TIMESTAMP = Thu Jan 2, 1997 11:28:03 AM
  NEWOWNER = Tom
DocumentChangeOwnerOverride: OBJECT = Document Checker 0 TYPE = Document
  NAME = Checker REVISION = 0 OWNER = Harry
ACCESSFLAG = True USER = creator
  CHECKACCESSFLAG = True
  TIMESTAMP = Thu Jan 2, 1997 11:28:04 AM
  NEWOWNER = Tom
DocumentChangeOwnerCheck: OBJECT = Document Checker 0 TYPE = Document
  NAME = Checker REVISION = 0 OWNER = Harry
ACCESSFLAG = True USER = creator
  CHECKACCESSFLAG = True
  TIMESTAMP = Thu Jan 2, 1997 11:28:05 AM
  NEWOWNER = Dick

```

Notice that the changeowner override program issues its own `modify businessobject Document EventTriggers 0 owner Dick` command. When Harry attempts to change the owner, this command causes a nested changeowner event to occur, which causes a second set of Triggers to fire. Note that the second Override program is a no-op due to the fact that the first override program is still running. Note also that since this second Override program is a no-op, the normal event activity (in this case, the first Override program) is guaranteed to take place. This is a very important point to keep in mind. So, at this point, the owner has been changed to Dick.

After the nested changeowner event transaction is committed, and the associated action program, if any, is run, control is returned to the original changeowner event. If the exit code from the override program is non-zero, the normal event activity is skipped and the owner name will continue to be Dick. However, if the exit code is zero, the normal activity takes place (overriding the owner change that took place in the nested event transaction). Now the owner name will be changed to Tom. In either case, the original event transaction is committed and the action program (if any) is then run.

Now let's look at the case of the changeowner Action program issuing its own `modify businessobject Document Checker 0 owner Harry` command. Assume that `RETURN_OVERRIDE_CODE=0` so that the event is not replaced.

```

#Changeowner event Action Program on Document Type
mql
code "output 'DocumentChangeOwnerAction: OBJECT = ${OBJECT} TYPE =${TYPE} ';
output ' NAME = ${NAME} REVISION = ${REVISION} OWNER = ${OWNER} ';
output ' USER = ${USER} ';
output ' TIMESTAMP = ${TIMESTAMP} ';
output ' NEWOWNER = ${NEWOWNER} ';
modify bus ${TYPE} ${NAME} ${REVISION} owner Harry;
quit;";

```

In this case, the command causes a new changeowner event to occur after the initial event transaction has been committed. When it is time for the Action Program to be run, the owner has changed to Tom. As before, recursion occurs and the second Action program is a no-op since the first action program is still running. Note that the second owner change

to Harry, which is caused by the Action program, will override the owner change to Tom that took place as the normal event transaction before the Action program was executed.

Now let's look what happens when both the changeowner override and action programs generate changeowner events as defined earlier. Assume that

RETURN_OVERRIDE_CODE=1 again, so that the event is replaced. In this case, we see the combined affects of both programs. The question is, "who now owns the Document object?" The only way to find out is to trace through the changes in a sequential fashion.

- 1.0) Original ChangeOwner event transaction is started. Check fires.
- 2.0) Override changes owner to Dick.
 - 2.1) Nested ChangeOwner transaction is started. Check fires.
 - 2.2) Override is a no-op, since the same Override Program (step 2.0) is still running. Transaction is committed.
 - 2.3) Nested event's Action changes owner to Harry.
 - 2.3.1) Nested ChangeOwner transaction is started. Check fires.
 - 2.3.2) Override is a no-op, since the same Override Program (step 2.0) is still running. Transaction is committed.
 - 2.4) Nested Action is a no-op, since the Action (step 2.3) is still running. Transaction is committed.
- 3.0) Original ChangeOwner event Action changes owner to Harry. (redundant).

With the original event transaction now committed, it seems as if the Action program changes will prevail. But let's continue.

- 3.1) ChangeOwner transaction is started. Check fires.
- 3.2) Override changes the owner to Dick.
 - 3.2.1) Nested Changeowner transaction is started. Check fires.
 - 3.2.2) Override is a no-op since the Override program (step 3.2) is still running. Transaction is committed.
 - 3.3.3) Nested Action is a no-op since the Action program (step 3.0) is still running. Transaction is committed.
- 3.3) Action is a no-op since the Action program (step 3.0) is still running. Transaction is committed.

So the proper conclusion to reach is that an Override program that replaces normal event activity will always win out, as should be the case. This example is simplistic in that the Check program was not involved, and the Override and Action programs issued single commands, which generated the same event. Obviously, Trigger programs will issue many commands and the events generated by these commands will trigger other programs, and so on. Because of this complexity, you may wish to graph the events, showing time on one axis, and the level of nesting on the other.

Logging Trigger Events

When designing and testing Triggers, it is highly recommended that a Log file is used so that when unexpected behavior occurs, it can be traced to the culprit program.

The eMatrix event log is enabled by setting the following environment variable:

```
MX_TRIGGER_LOG=TRUE
```

When the above line is added to the initialization file or the UNIX startup scripts, eMatrix will create a file called `mxtrigger.log` in the `MATRIXHOME` directory. The log will contain the following information for each trigger that is fired during that session.

TIMESTAMP	EVENT	TRIGGER-TYPE	PROGRAM NAME
-----------	-------	--------------	--------------

For example:

```
Mon, May 12, 1997 09:32 EST    modifyattribute    action Update    Provider Range
```

Subsequent sessions will append entries to the existing file, provided the log is still enabled.

MQL trace can also be used to test timing and debug trigger programs. For information, see [Testing Timing and Debugging Programs](#) in Chapter 19.

Enable/Disable Triggers

Triggers can be disabled for a session, without having to explicitly remove them from the administrative definitions to which they belong. This is helpful during testing and troubleshooting.

The following command can be executed from MQL provided that the current user is defined as a System Administrator.

```
trigger off;
```

After this command is executed, events that occur on the local machine which have associated triggers do not cause the programs to be executed.

This command affects only the local machine; concurrent user's sessions are not affected. Even when multiple users are sharing one executable, each user is isolated from all other users. In addition, subsequent sessions on the local machine will have triggers enabled by default.

To enable defined triggers again, a System Administrator should use:

```
trigger on;
```

MQL also allows triggers to be enabled or disabled with the use of a toggle command. Depending on the current setting, triggers can be enabled/disabled using the same command.

To enable/disable triggers as a toggle, a System Administrator should use:

```
trigger;
```

21

Working With Workflow Processes

Overview of Workflow Processes	542
Defining a Process	543
Description Clause.....	543
Attribute Clause	544
Interactive Clause	544
Automated Clause	548
Subprocess Clause.....	551
And Clause	552
Or Clause.....	552
Finish Clause.....	552
Start Clause	552
Stop Clause	553
Autostart Clause	553
Hidden Clause	553
Icon Clause.....	553
Property Clause	553
Trigger Clause	554
Copying and/or Modifying a Process Definition.....	555
Copying (Cloning) a Process Definition	555
Modifying a Process Definition	555
Reassigning an Activity to a Group	558
Using Links	559
Connect Clause	559
Disconnect Clause.....	559
Transition Conditions	559
Deleting a Process.....	561
Validating a Process.....	562

Overview of Workflow Processes

Workflow is concerned with the automation of procedures, where documents, information or tasks are passed among participants according to a defined set of rules to achieve or contribute to an overall business goal.

Business Administrators create Workflow *Process* definitions. The process definition contains a set of activities (tasks) connected with intelligent links that allow branching within the process. The process can also include previously-defined sub-processes.

Any user can create Workflow *Instances*, based on Processes. When a workflow is launched, the workflow instance and all the constituent activity instances are automatically created. The task assignments are dropped off in each user's IconMail inbox. A user completes the task and communicates the status of that task assignment to the workflow system. Based on the rules defined in the process definition, the workflow system routes the task to the next task performer(s), until the workflow is completed. Workflow owners are allowed to abort or suspend a workflow instance. (See *Overview of Workflows* in Chapter 33.)

Defining a Process

A process is created with the Add Process statement:

```
add process NAME [ADD_ITEM {ADD_ITEM}];
```

NAME is the name you assign to the process. The process name is limited to 127 characters. The naming convention for process objects is similar to conventions for business objects. For additional information, refer to *Business Object Name* in Chapter 34.

ADD_ITEM is an Add Process clause that provides additional information about the process. The Add Process clauses are:

description STRING
attribute NAME { ,NAME }
interactive ACTIVITY_NAME [ACTIVITY_ITEM { ,ACTIVITY_ITEM }]
automated AUTO_ACTIVITY_NAME [AUTO_ACTIVITY_ITEM { ,AUTO_ACTIVITY_ITEM }]
subprocess SUBPROCESS_NAME reference PROCESS_NAME [description STRING] [xcoord VALUE] [ycoord VALUE]
and AND_NODE_NAME
or OR_NODE_NAME
finish FINISH_NODE_NAME
start START_NODE_NAME
stop STOP_NODE_NAME
autostart on off
[! not]hidden
icon FILENAME
property NAME [to ADMINTYPE NAME] [value STRING]
TRIGGER PROG_NAME [input ARG_STRING]

All these clauses are optional. You can define a process by simply assigning a name to it. (Note, however, that this skeleton process cannot be instantiated as an eMatrix workflow since it lacks required process elements.) Each Add Process clause is described in detail in the sections that follow.

Description Clause

The Description clause of the Add Process statement provides general information for you and the user about the process and the overall function of the process. There may be subtle differences between processes; the description can point out the differences. The syntax of the Description clause is:

```
description STRING
```

STRING is a string of any length, enclosed in quotes if it contains embedded spaces.

There is no limit to the number of characters you can include in the description. However, keep in mind that the description is displayed when the mouse pointer stops over the process in the Process chooser. Although you are not required to provide a description, this information is helpful when a choice is requested.

Attribute Clause

The Attribute clause of the Add Process statement assigns explicit attributes to the process. These attributes must be previously defined with the Add Attribute statement or in Business Modeler. If they are not defined, an error message is displayed.

If attributes are defined for the process, the `attribute` clause must precede any interactive or automated clauses in the Add Process statement.

In addition to adding attributes to a process as a whole, attributes can also be added to the activities within a process. This is similar to how attributes are used within a type hierarchy where an attribute can be added for a parent type and then additional attributes can be added to the child type of that parent.

For the user, the display of attributes (when creating processes or viewing attributes) will appear in the reverse order of the programmed order. Therefore, you should put the first attribute last in the MQL script.

A process can have any combination of attributes associated with it. For example, the following Add Process statement assigns three attributes to the Create Part process:

```
add process "Order Materials"
    description "to purchase materials for manufacturing"
    attribute "Total Cost"
    attribute "Quantity"
    attribute "Composition";
```

Interactive Clause

The Interactive clause of the Add Process statement adds an interactive activity to the process.

During workflow execution, interactive activities are performed by a workflow participant who becomes the owner of that activity. The activity owner may need to modify attributes of the activity and may also need to access existing attachments or add business objects as attachments.

The Interactive clause uses the following syntax:

```
interactive ACTIVITY_NAME [ACTIVITY_ITEM { ,ACTIVITY_ITEM} ]
```

`ACTIVITY_NAME` is the name assigned to the interactive activity.

`ACTIVITY_ITEM` is a subclause of the Interactive clause that defines the interactive activity. The Interactive subclauses are:

name NAME
description STRING
attribute NAME { ,NAME}
user ASSIGNEE
duration VALUE

wizard NAME
priority VALUE
instruction VALUE
rate RATE
xcoord VALUE
ycoord VALUE
trigger EVENT_TYPE TRIGGER_TYPE PROG_NAME [input ARG_STRING]

All these subclauses are optional. You can define an interactive activity simply by assigning a name to it. Each subclause is described in detail in the sections that follow.

Description subclause

The Description subclause of the Interactive clause provides general information about the interactive activity and the overall function of the activity. The syntax of the Description subclause is:

description STRING

STRING is a string of any length, enclosed in quotes if it contains embedded spaces.

Attribute subclause

The Attribute clause of the Interactive clause assigns explicit attributes to the interactive activity. These attributes must be previously defined with the Add Attribute statement or in Business Modeler. If they are not defined, an error message is displayed.

For the user, the display of attributes (when creating processes or viewing attributes) will appear in the reverse order of the programmed order. Therefore, you should put the first attribute last in the MQL script.

An interactive activity can have any combination of attributes associated with it. For example, the following Add Process statement assigns the attribute TotalCost to the interactive activity:

add process "Create Part" description "to purchase parts for manufacturing process" interactive "Order Materials" attribute "TotalCost";

User subclause

The User subclause of the Interactive clause adds an assigned user to the interactive activity. The Process definition must contain assignees for all interactive activities. All users who are assigned workflow tasks must have IconMail enabled. See [Enable Iconmail Clause](#) in Chapter 11. The syntax of the User subclause is:

user ASSIGNEE

ASSIGNEE is a person, group, role or association defined in eMatrix.

For example, the following statement assigns the interactive activity “Check References” to group “Human Resources.”

```
add process "Hiring"
    interactive "Check References"
        user "Human Resources";
```

Duration subclause

The Duration subclause of the Interactive clause specifies the number of days allocated for the interactive activity. The syntax of the Duration subclause is:

```
duration VALUE
```

VALUE can be any whole number.

For example, the following statement allocates 3 days to the interactive activity “Update Chapter 3.”

```
add process "Update Book"
    interactive "Update Chapter 3"
        duration 3;
```

Wizard subclause

The Wizard subclause of the Interactive clause allows you to add Wizards or other programs to a workflow interactive activity definition to make it easier for workflow participants to perform tasks. Wizards can be designed to facilitate any task within the workflow. You can create and assign Wizards such as Complete, Reassign, Suspend, etc. to the activity definition, which the end users would execute to communicate to the workflow system. Any program or tool that you specify appears as a toolbar icon within the task window when the task is opened in eMatrix. You can add multiple tools within a single Interactive task. These wizards/programs must be previously defined with the Add Wizard statement or in Business Modeler. If they are not defined, an error message is displayed.

The syntax of the Wizard subclause is:

```
wizard NAME
```

NAME is the name of the wizard/program as it exists in the database.

Priority subclause

The Priority subclause of the Interactive clause helps to prioritize the tasks to be performed (applicable when a user has more than one task assigned).

The syntax of the priority subclause is:

```
priority VALUE
```

VALUE can be one of the following: *urgent*, *high*, *medium*, *low*.

Instruction subclause

The Instruction subclause of the Interactive clause specifies instructions for the assigned user of the activity. The syntax for the Instruction subclause is:

```
instruction VALUE
```

VALUE can be a string of any length describing the work that needs to be done to complete the activity. For example, if the activity involves purchase requisitions, and they must be entered each week before 3 p.m. on Thursday afternoon, this information can be included in the Instruction subclause. For example:

```
add process "Purchase Req"
    interactive "Enter reqs"
    instruction "All reqs must be entered into the
    database before 3 p.m. on Thurs. afternoon.;"
```

Rate subclause

The Rate subclause of the Interactive clause specifies the percentage of the assigned user's time expected to be allocated for this activity. The syntax of the Rate subclause is:

```
rate RATE
```

RATE can be a decimal number representing a percentage of the whole.

For example, if you expect the "Update Chapter 3" activity to take 5 days when the assigned user is working on this activity for half of every day, you would use the following statement:

```
add process "Update Book"
    interactive "Update Chapter 3"
    user "lois"
    duration 5
    rate 0.50;
```

Xcoord subclause

The Xcoord subclause of the Interactive clause specifies horizontal location on the graph of the Interactive activity icon. The syntax of the Xcoord subclause is:

```
xcoord VALUE
```

VALUE must be a whole number.

Ycoord subclause

The Ycoord subclause of the Interactive clause specifies vertical location on the graph of the Interactive activity icon. The syntax of the Ycoord subclause is:

```
ycoord VALUE
```

VALUE must be a whole number.

Trigger subclause

Triggers allow the execution of a Program object to be associated with the occurrence of an event.

Interactive activity Triggers use the following syntax:

```
trigger EVENT_TYPE TRIGGER_TYPE PROG_NAME [input ARG_STRING]
```

EVENT_TYPE is any of the valid events for interactive activities:

activateactivity, completeactivity, suspendactivity,
resumeactivity, overrideactivity, reassignactivity

TRIGGER_TYPE is check, override, or action. Refer to [Types of Triggers](#) in Chapter 20.

PROG_NAME is the name of the Program object that will execute when the event occurs.

ARG_STRING is a string of arguments to be passed into the program. When you pass arguments into the program they are referenced by variables within the program.

Variables 0, 1, 2... etc. are reserved by the system for passing in arguments.

Environment variable “0” always holds the program name and is set automatically by the system.

Arguments following the program name are set in environment variables “1”, “2”,... etc.

See [Using the Runtime Program Environment](#) in Chapter 19 for additional information on program environment variables.

For example:

```
add process "Create Part"  
    interactive "Machine Part"  
    trigger startprocess action "Record Starttime";
```

Refer to [Designing Triggers](#) in Chapter 20, for more information on Triggers.

Automated Clause

The Automated clause of the Add Process statement adds an automated activity to the process.

During workflow execution, automated activities are directly activated by the workflow system with no workflow participant or task performer involved. A program object is assigned to an automatic activity definition. This program object is executed at the appropriate time during execution phase.

The Automated clause uses the following syntax:

```
automated AUTO_ACTIVITY_NAME [AUTO_ACTIVITY_ITEM  
{,AUTO_ACTIVITY_ITEM}];
```

AUTO_ACTIVITY_NAME is the name assigned to the automated activity.

AUTO_ACTIVITY_ITEM is a subclause of the Automated clause that defines the automated activity. The Automated subclauses are:

```
name NAME
```

```
description STRING
```

attribute NAME { ,NAME }
user ASSIGNEE
xcoord XVALUE
ycoord YVALUE
program PROG_NAME [input ARG_STRING]
trigger EVENT_TYPE TRIGGER_TYPE PROG_NAME [input ARG_STRING]

All these subclauses are optional. You can define an automated activity simply by assigning a name to it. Each subclause is described in detail in the sections that follow.

Description subclause

The Description subclause of the Automated clause provides general information about the automated activity and the overall function of the activity. The syntax of the Description subclause is:

description STRING

STRING is a string of any length, enclosed in quotes if it contains embedded spaces.

Attribute subclause

The Attribute clause of the Automated clause assigns explicit attributes to the automated activity. These attributes must be previously defined with the Add Attribute statement or in Business Modeler. If they are not defined, an error message is displayed.

For the user, the display of attributes (when creating processes or viewing attributes) will appear in the reverse order of the programmed order. Therefore, you should put the first attribute last in the MQL script.

An automated activity can have any combination of attributes associated with it. For example, the following Add Process statement assigns the attribute "Time" to the "Log Progress" automated activity:

add process "Create Part" automated "Log Progress" attribute "Time";
--

User subclause

The User subclause of the Automated clause adds an assigned user to the automated activity. The syntax of the User subclause is:

user ASSIGNEE

ASSIGNEE is a person, group, role or association defined in eMatrix.

If the user subclause is not defined, the automated activity will be performed at the appropriate time “in place,” that is, on the machine where the task which triggered its activation was run.

You may, however, require that the automated activity be executed on a particular PC. This PC might have necessary application software, special integrations, etc. For example,

you may need to access the finance database which is not available to all users. In this case, you should specify an assigned user whose context has access to that database. To prevent the task assignment from showing in the person's inbox, you could create an alternate name for the user and use that name instead.

Any automated activity that has an assignee must be executed through a cron/batch program which can be set up to run on any machine where resources are available to execute the program object attached to the automated activity. The batch program can be an MQL script which would set context as the assignee/special agent and run the execute command. For example:

```
execute workflow PROCESS NAME automated AUTO_NAME;
```

PROCESS is the name of the process on which the workflow is based.

NAME is the name of the workflow you want to start.

AUTO_NAME is the name of the automated activity that you want to execute.

Xcoord subclause

The Xcoord subclause of the Automated clause specifies horizontal location on the graph of the automated activity icon. The syntax of the Xcoord subclause is:

```
xcoord VALUE
```

VALUE must be a whole number.

Ycoord subclause

The Ycoord subclause of the Automated clause specifies vertical location on the graph of the automated activity icon. The syntax of the Ycoord subclause is:

```
ycoord VALUE
```

VALUE must be a whole number.

Program subclause

The Program subclause of the Automated clause adds the program which executes when the automated activity is started. The syntax of the Program subclause is:

```
program PROG_NAME [input ARG_STRING]
```

PROG_NAME is the name of the program.

ARG_STRING is a string of arguments to be passed into the program. When you pass arguments into the program they are referenced by variables within the program. Variables 0, 1, 2... etc. are reserved by the system for passing in arguments.

Environment variable "0" always holds the program name and is set automatically by the system.

Arguments following the program name are set in environment variables "1", "2", ... etc.

See [Using the Runtime Program Environment](#) in Chapter 19 for details.

Trigger subclause

Triggers allow the execution of a Program object to be associated with the occurrence of an event.

Automated activity Triggers use the following syntax:

```
trigger EVENT_TYPE TRIGGER_TYPE PROG_NAME [input ARG_STRING]
```

EVENT_TYPE is any of the valid events for automated activities:
activateautoactivity, completeautoactivity,
suspendautoactivity, resumeautoactivity,
overrideautoactivity, reassigntoautoactivity

TRIGGER_TYPE is check, override, or action. Refer to [Types of Triggers](#) in Chapter 20.

PROG_NAME is the name of the Program object that will execute when the event occurs.

ARG_STRING is a string of arguments to be passed into the program. When you pass arguments into the program they are referenced by variables within the program.
Variables 0, 1, 2... etc. are reserved by the system for passing in arguments.

Environment variable “0” always holds the program name and is set automatically by the system.

Arguments following the program name are set in environment variables “1”, “2”,... etc.

See *Runtime Program Environment* in the *Programs* chapter for additional information on program environment variables.

For example:

```
add process "Create Part"
    interactive "Machine Part"
    trigger startprocess action "Record Starttime";
```

Refer to [Designing Triggers](#) in Chapter 20 for more information on Triggers.

Subprocess Clause

The Subprocess clause of the Add Process statement adds a sub-process to the process.

A sub-process is a process that is enacted or called from another process and forms part of the overall process. A sub-process is useful for defining reusable components from within other processes. A sub-process will have its own process definition.

The sub-process informs workflow in which it is contained (the parent process) that it is complete when the sub-process reaches the Finish disposition.

The Subprocess clause uses the following syntax:

```
subprocess SUBPROCESS_NAME reference PROCESS_NAME description
    STRING] [xcoord XVALUE] [ycoord YVALUE]
```

SUBPROCESS_NAME is the name assigned to the sub-process.

PROCESS_NAME is the originally defined process that is being referenced as a sub-process.

STRING is a string of any length, enclosed in quotes if it contains embedded spaces, which provides general information about the sub-process and the overall function of the sub-process.

XVALUE is a whole number that specifies the horizontal location on the graph of the sub-process icon.

YVALUE is a whole number that specifies the vertical location on the graph of the sub-process icon.

And Clause

The And clause of the Add Process statement adds an AND connector to the process. AND connectors can be added to a process to create branching within the process. For example, if your process defines the activities necessary to create a book, you may need different translations of the book to be performed at the same time. The syntax of the And clause is:

```
and AND_NODE_NAME
```

AND_NODE_NAME is the name assigned to the AND connector. This name is used in the Modify Process statement when linking activities and sub-processes through an AND connector.

Or Clause

The Or clause of the Add Process statement adds an OR connector to the process. OR connectors can be added to a process to create activity branching within the process. For example, if your process defines the activities necessary to approve a purchase requisition, you may need different approvals depending on the cost of the item. The syntax of the Or clause is:

```
or OR_NODE_NAME
```

OR_NODE_NAME is the name assigned to the OR connector. This name is used in the Modify Process statement when linking activities and sub-processes through an OR connector.

Finish Clause

The Finish clause of the Add Process statement adds a finish node to the process, which signifies the end of the process. The Finish node is required, and there can be only one Finish node in a process definition. The syntax of the Finish subclause is:

```
finish FINISH_NODE_NAME
```

FINISH_NODE_NAME is the name assigned to the Finish node. This name is used in the Modify Process statement when linking the final activity or sub-process to the Finish node.

Start Clause

The Start clause of the Add Process statement adds a start node to the process, which signifies the start of the process. There can be only one Start node in a process definition. The syntax of the Start subclause is:

```
start START_NODE_NAME
```

START_NODE_NAME is the name assigned to the Start node. This name is used in the Modify Process statement when linking the first activity or sub-process to the Start node.

Stop Clause

The Stop clause of the Add Process statement adds a stop node to the process, which signifies an abrupt termination of the process. The syntax of the Stop subclause is:

```
stop STOP_NODE_NAME
```

STOP_NODE_NAME is the name assigned to the Stop node. This name is used in the Modify Process statement when linking an activity or sub-process to the Stop node.

Autostart Clause

The Autostart clause of the Add Process statement specifies whether the workflow is started as soon as it is created in eMatrix Navigator. If you want the workflow to start immediately after it is created, use:

```
autostart on
```

The default is autostart off.

Hidden Clause

You can specify that the new process is “hidden” so it does not appear in the Process chooser in eMatrix, which simplifies the end-user interface. Users who are aware of the hidden process’s existence can enter its name manually where appropriate. Hidden objects are also accessible through MQL.

The hidden flag can be changed even when instances of the workflow are active. This allows Business Administrators to clone existing process definitions and modify the clones to the new process definition. Marking the old process as hidden would force eMatrix Navigator users to use the new process definitions.

Icon Clause

The Icon clause of the Add Process statement associates a special image with a process. Icons help users locate and recognize items. You can assign a special icon to the new process or use the default icon. The default icon is used when in view-by-icon mode. Any special icon you assign is used when in view-by-image mode. When assigning a unique icon, you must use a GIF image file. Refer to [Icon Clause](#) in Chapter 1 for a complete description of the Icon clause.

GIF filenames should not include the @ sign, as that is used internally by eMatrix.

Property Clause

Integrators can assign ad hoc attributes, called Properties, to the process. Properties allow associations to exist between administrative definitions that aren’t already associated. The property information can include a name, an arbitrary string value, and a reference to another administration object. The property name is always required. The value string and object reference are both optional. The property name can be reused for different object references, that is, the name joined with the object reference must be unique for any object that has properties.

```
add process NAME  
property NAME [to ADMINTYPE NAME] [value STRING];
```

For additional information on properties, see *Overview of Administration Properties* in Chapter 24.

Trigger Clause

Triggers allow the execution of a Program object to be associated with the occurrence of an event.

Process Triggers use the following syntax:

```
trigger EVENT_TYPE TRIGGER_TYPE PROG_NAME [input ARG_STRING];
```

EVENT_TYPE is any of the valid events for Processes: startprocess, finishprocess, suspendprocess, resumeprocess, stopprocess, reassignprocess.

TRIGGER_TYPE is check, override, or action. Refer to [Types of Triggers](#) in Chapter 20.

PROG_NAME is the name of the Program object that will execute when the event occurs.

ARG_STRING is a string of arguments to be passed into the program. When you pass arguments into the program they are referenced by variables within the program. Variables 0, 1, 2... etc. are reserved by the system for passing in arguments.

Environment variable “0” always holds the program name and is set automatically by the system.

Arguments following the program name are set in environment variables “1”, “2”,... etc.

See [Using the Runtime Program Environment](#) in Chapter 19 for additional information on program environment variables.

For example:

```
add process  
    trigger startprocess action "Schedule workflow";
```

Refer to [Designing Triggers](#) in Chapter 20 for more information on designing Triggers.

Workflow Macros

Macros are a simple name/value string substitution mechanism, where the macro value is substituted for the macro name when used in a Program as follows:

```
 ${MACRONAME}
```

This single one-pass string substitution process occurs just prior to program execution. These macros are also created as a variable of the same name in the RPE. Refer to the [Macros Appendix](#) for more information.

Copying and/or Modifying a Process Definition

Copying (Cloning) a Process Definition

After a process is defined, you can clone the definition with the Copy Process statement. This statement lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy process SRC_NAME DST_NAME [MOD_ITEM {MOD_ITEM}] ;
```

SRC_NAME is the name of the process definition (source) to copied.

DST_NAME is the name of the new definition (destination).

MOD_ITEMS are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

Modifying a Process Definition

After a process is defined, you can change the definition with the Modify Process statement.

The following statement lets you add or remove defining clauses and change the value of clause arguments:

```
modify process NAME [MOD_ITEM {MOD_ITEM}] ;
```

NAME is the name of the process you want to modify.

MOD_ITEM is the type of modification you want to make.

You can make the following modifications. Each is specified in a Modify Process clause, as listed in the following table. Note that you only need to specify fields to be modified.

Modify Process Clause	Specifies that...
name NAME	The current name is changed to the name entered.
description VALUE	The current description, if any, changes to the value entered.
interactive ACTIVITY_NAME [ACTIVITY_ITEM { ,ACTIVITY_ITEM}]	The named interactive activity is modified.
automated AUTO_ACTIVITY_NAME [AUTO_ACTIVITY_ITEM { ,AUTO_ACTIVITY_ITEM}]	The named automated activity is modified.
subprocess SUBPROCESS_NAME reference PROCESS_NAME [description VALUE] [xcoord XVALUE] [ycoord YVALUE]	The named sub-process is modified.
connect from NODE_NAME to NODE_NAME	The elements (interactive activities, automated activities, sub-processes, etc.) specified by NODE_NAME are connected by a link.
disconnect from NODE_NAME to NODE_NAME	The link between the elements (interactive activities, automated activities, sub-processes, etc.) specified by NODE_NAME is disconnected.

Modify Process Clause	Specifies that...
from FROM_NODE to TO_NODE transition expression EXPRESSION	An expression is added to the specified link, or the expression on the specified link is changed to the new expression. FROM_NODE and TO_NODE define the elements on either end of the link.
icon FILENAME	The image is changed to the new image in the file specified.
hidden	The hidden option is changed to specify that the object is hidden.
nothidden	The hidden option is changed to specify that the object is not hidden.
property NAME [to ADMINTYPE NAME] [value STRING]	The named property is modified.
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.
add attribute NAME	The named attribute is added.
remove attribute NAME	The named attribute is removed.
add interactive ACTIVITY_NAME [ACTIVITY_ITEM {,ACTIVITY_ITEM}]	The named interactive activity is added.
remove interactive ACTIVITY_NAME [ACTIVITY_ITEM {,ACTIVITY_ITEM}]	The named interactive activity is removed. If ACTIVITY_ITEMS are specified, only those items are removed from the activity; the interactive activity itself is not removed. All ACTIVITY_ITEMS can be removed except for Name, Description, Duration, Priority, and Rate.
add automated AUTO_ACTIVITY_NAME [AUTO_ACTIVITY_ITEM {,AUTO_ACTIVITY_ITEM}]	The named automated activity is added.
remove automated AUTO_ACTIVITY_NAME [AUTO_ACTIVITY_ITEM {,AUTO_ACTIVITY_ITEM}]	The named automated activity is removed. If ACTIVITY_ITEMS are specified, only those items are removed from the activity; the automated activity itself is not removed. All ACTIVITY_ITEMS can be removed except for Name and Description.
add subprocess SUBPROCESS_NAME [description VALUE] [xcoord XVALUE] [ycoord YVALUE]	The named sub-process is added.
add or OR_NODE_NAME	The named OR node is added.
remove or OR_NODE_NAME	The named OR node is removed.
add and AND_NODE_NAME	The named AND node is added.
remove and AND_NODE_NAME	The named AND node is removed.
add finish FINISH_NODE_NAME	The named finish node is added.
remove finish FINISH_NODE_NAME	The named finish node is removed.
add start START_NODE_NAME	The named start node is added.

Modify Process Clause	Specifies that...
remove start START_NODE_NAME	The named start node is removed.
add stop STOP_NODE_NAME	The named stop node is added.
remove stop STOP_NODE_NAME	The named stop node is removed.
add from FROM_NODE to TO_NODE transition EXPRESSION	The expression is added to the link between nodes identified by FROM_NODE and TO_NODE.
remove from FROM_NODE to TO_NODE transition EXPRESSION	The expression is removed from the link between nodes identified by FROM_NODE and TO_NODE.
add trigger EVENT_TYPE TRIGGER_TYPE PROG_NAME	The specified trigger type is added to the listed event.
remove trigger EVENT_TYPE TRIGGER_TYPE PROG_NAME	The specified trigger type is removed from the listed event.

Each modification clause is related to the arguments that define the process.

Reassigning an Activity to a Group

Any workflow activity can be reassigned in an instantiated workflow by its owner as long as the activity has not been completed. The new owner can be any type of user (person, group, role, or association). You can reassign activities before the workflow has been started, while it is in progress, or by stopping (and restarting) it. (You cannot reassign activities when the workflow has been suspended or completed.) When you restart a workflow, it returns to the beginning of the process.

There is a behavior difference between assigning an activity to a group in the process definition and reassigning an activity to a group in the workflow instance:

- If the assignee of an activity in a process definition is a group or role, the workflow engine routes a task to each member of the group, or each person assigned to the role. Any of the recipients of the taskmail can then accept the task (from eMatrix or MQL), and become the owner of the activity. The taskmail is then rescinded from all other group members' inboxes.
- If an activity is reassigned to a group or role from the workflow instance, the workflow engine routes a task to each member of the group, or each person assigned to the role. The activity does not require acceptance, and it remains under the ownership of the group or role until it is completed. No taskmails are rescinded until the task is completed, and the user that completes the task is recorded in history.

When members of a group receive taskmail for an activity that has been reassigned to the group, any member can work on the activity. The taskmail remains in all members' inboxes as a reminder of the group's responsibility. When a member marks the task complete, all taskmails for this activity are rescinded from all members.

Using Links

The elements of a process definition are linked to each other to form a directed process. Links are created using the Connect clause of the Modify Process statement. Links must be made between each of the elements that have been defined for the process, including interactive activities, automated activities, sub-processes, AND-connectors, OR-connectors, start node, finish node, and stop node(s).

Connect Clause

After elements of a process have been defined, you must link them using the Connect clause of the Modify Process statement. The syntax is:

```
modify process connect from NODE_NAME to NODE_NAME;
```

NODE_NAME is a name of a previously defined element of the process.

For example, to connect a start node named StartChk to the first activity definition named Assign Number, use the following:

```
modify process connect from StartChk to "Assign Number";
```

Disconnect Clause

To disconnect elements of a process that have been connected, use the Disconnect clause of the Modify Process statement. The syntax is:

```
modify process disconnect from FROM_NODE to TO_NODE;
```

FROM_NODE is the name of the node at the start of the link that you want to disconnect.

TO_NODE is the name of the node at the end of the link that you want to disconnect.

All links connected to a node should be disconnected before replacing/removing any nodes.

Transition Conditions

A transition condition is a logical expression to be evaluated by the workflow system. This decides the sequence of activity execution within a workflow. It is stored as an attribute of the link. The transition condition is used with the OR clause, where multiple alternative workflow branches exist and the system needs to determine which branch to take to advance the workflow. This approach allows adding as many alternative branches on the OR icon as needed without cluttering the activity instance with branch conditions for each branch that is needed.

Transition conditions are added using the following clause of the Modify Process statement:

```
add from FROM_NODE to TO_NODE transition expression EXPRESSION
```

FROM_NAME is a name of the node at the “from” side of the link.

TO_NAME is a name of the node at the “to” side of the link.

EXPRESSION is the expression to be evaluated.

For example, the following expression could be used to route tasks of cost value less than or equal to 25:

```
modify process "Order Supplies"  
add from OR1 to "Place Order" transition expression  
"interactive[Task1].attribute[Actual Cost].value <= 25";
```

See [Where Clause](#) in Chapter 38 for information on how to create logical expressions.

Processes should be designed so that only one transition condition can be true for each OR-split. For example, consider the following transition conditions:

```
interactive[Task1].attribute[Actual Cost].value <= 25
```

and

```
interactive[Task1].attribute[Actual Cost].value >= 25
```

The first tests for Actual Cost less than or equal to 25. The second tests for Actual Cost greater than or equal to 25. If the Actual Cost is exactly 25, both conditions test true. The workflow will take the first path it finds that tests true. Since the transition conditions are ambiguous, you may get unexpected results.

Business object attributes can also be used in transition conditions. These business objects need to be used as attachments before evaluation can be done. For example:

```
businessobject[TYPE NAME REV].attribute[ATTR_NAME].value == 25
```

```
businessobject[ECR 2000 0].attribute[Type of Change].value == "ELECTRICAL"
```

It is not necessary to add transition conditions to every link in a process. If a link does not contain a transition expression, it provides a default path. If none of the transition conditions for a particular OR-split is met, the task on the default path would be activated.

Modifying Transition Conditions

Transition conditions can be modified using the following clause of the Modify Process statement:

```
from FROM_NODE to TO_NODE transition expression EXPRESSION
```

FROM_NAME is a name of the node at the “from” side of the link.

TO_NAME is a name of the node at the “to” side of the link.

EXPRESSION is the new expression to be evaluated. It replaces whatever expression was previously defined on the link between the named nodes.

Deleting a Process

If a process is no longer required, you can delete it with the Delete Process statement:

```
delete process NAME;
```

NAME is the name of the process to be deleted.

When this statement is executed, eMatrix searches the list of processes. If the name is not found, an error message is displayed. If the name is found, the process is deleted.

Validating a Process

When eMatrix attempts to create or modify a defined process, it checks for a couple of errors in the process definition. If eMatrix finds one of these errors, it presents an error message and will not save the process:

- A sub-process is included that has not been defined.
- Invalid links between process elements.

If you don't receive an error message when you save a process definition, the process may still have errors. eMatrix allows you to save a process that has some errors so you can save processes that aren't complete. Here is a list of errors that may be present in a process but that will not prevent the system from saving the process:

- Missing or more than one Start node
- Start node not at the beginning of the process
- More than one node connected to the Start node
- Missing or more than one Finish node
- Finish node not at the end of the process
- Stop node with nodes following it
- No activities or sub-processes included in the process
- No assignee for interactive activities
- An AND connector following an OR connector
- An AND connector following an AND connector

To check for the above errors, you can validate a process using the Validate Process statement:

```
validate process PROCESS_NAME { ,PROCESS_NAME} ;
```

PROCESS_NAME is the name of the process to be validated.

When this statement is executed, eMatrix checks the process and ensures that all validation checks are met.

22

Working With Reports

Overview of Reports	564
Designing the Definition and Layout of a Report	565
Defining a Report.....	566
Units Clause	566
Description Clause.....	567
Icon Clause.....	568
Size Clause.....	568
Header Clause.....	568
Footer Clause	569
Margins Clause.....	570
Displayrule Clause.....	571
Field Clause.....	571
Hidden Clause	580
Property Clause	580
Evaluating a Report	581
Copying and/or Modifying a Report.....	582
Copying (Cloning) a Report Definition	582
Modifying a Report.....	582
Deleting a Report	584

Overview of Reports

A *report* is an organized presentation of selected contents of business objects. Users can generate a report about selected objects using one of the report formats designed by you. The form determines:

- The type of information that is reported.
For example, a user might generate a Material Properties report for several assemblies. Based on the report definition, eMatrix collects certain types of information about the objects: name, description, material type, and target actual cost. The report might also reflect lifecycle approvals.
- The layout of the report.
The Material Properties report might list the information in four columns.

A prerequisite to using the report feature is that its base type is defined as described in *Type Defined* in Chapter 13. Other report types can be derived from this base type so that when a report is generated, a new Report object is created with a text file of the actual run report checked into it. When you create a report, you select the Report (format) and business object Type that will be created from the selected objects.

Report formats are dependent upon the types of business objects for which they are designed. For example, running a report which shows the total cost of a project by adding up the values for the attribute “Total Cost” will have no result if run on objects which do not have that attribute. Therefore, when evaluating reports, it is important to know and choose the appropriate report format to produce the desired output.

You must be a Business Administrator to define a report.

Designing the Definition and Layout of a Report

Business objects can contain a large amount of information, not all of which will be relevant to your report. Most reports are designed to answer specific questions. For example:

- What is the total cost of this project?
- Who is working on this project and how can they be contacted?

Once you have identified each question, you can determine the information required to provide each answer. To be efficient, a report should be designed to logically answer these questions. Simply seeing rows of names or numbers has little meaning to the reader. All information should be labeled and grouped appropriately so that a reader can easily locate desired values.

Defining a Report

Use the Add Report statement to define a report:

```
add report NAME [ADD_ITEM {ADD_ITEM}] ;
```

NAME is the name you assign to the report you are creating. The report name is limited to 127 characters. For additional information, refer to *Business Object Name* in Chapter 34.

ADD_ITEM is an Add Report clause which provides additional information about the report. The Add Report clauses are:

units [picas points inches]
description VALUE
icon FILENAME
size ROW_SIZE COL_SIZE
header HEADER_SIZE
footer FOOTER_SIZE
margins LEFT_MARGIN RIGHT_MARGIN
displayrule false true
field FIELD_TYPE FIELD_DEF {FIELD_DEF}
[! not]hidden
property NAME [to ADMINTYPE NAME] [value STRING]

With the exception of the Units and Size clauses, all other Add Report clauses are optional. (Units will default to picas if you do not enter a value.) Field clauses specify the field values that should be printed in the report and where. Without at least one Field clause, your report will not have much value.

In the sections that follow, you will learn more about each Add Report clause.

Units Clause

The Units clause of the Add Report statement specifies the units of page measurement. There are three possible values: picas, points, or inches.

units picas <i>Or</i> units points <i>Or</i> units inches

Without a unit of measurement, eMatrix cannot interpret the values of any given header, footer, margin, or field size. Because picas are the default unit of measurement, eMatrix will automatically assume a picas value if you do not use a Units clause.

Picas are the most common units of page measurement in the computer industry. Picas use a fixed size for all characters. Determining the size of a field value is easy when using picas as the measurement unit. Simply determine the maximum number of characters that

will be used to contain the largest field value. Use that value as your field size. For example, if the largest field value will be a six digit number, you need a field size of six picas. This is not true when using points.

Points are standard units used in the graphics and printing industry. A point is equal to 1/72 of an inch or 72 points to the inch. Points are commonly associated with fonts whose print size and spacing varies from character to character. Unless you are accustomed to working with points, measuring with points can be confusing and complicated. For example, the character “I” may not occupy the same amount of space as the characters “E” or “O.” To determine the maximum field size, you need to know the maximum number of characters that will be used and the maximum amount of space required to express the largest character. Multiply these two numbers to determine your field size value.

Inches are common English units of measurement. While you can use inches as your unit of measurement, be aware that field placement can be difficult to determine and specify. Each field is composed of character string values. How many inches does each character need or use? If the value is a four-digit number, how many inches wide must the field be to contain the value? How many of these fields can you fit across a report page? Considering the problems involved in answering these questions, you can see why picas are a favorite measuring unit.

Description Clause

The Description clause of the Add Report statement provides general information to you and the user about the function of the report. There may be subtle differences between reports; you can use the Description clause to point out the differences to the user.

There is no limit to the number of characters you can include in the description. However, keep in mind that the description is displayed when the mouse pointer stops over the report in a chooser. Although you are not required to provide a description, this information is helpful when a choice is requested.

You can distinguish your report in your selection of a report name. This consists of a character string value to identify the report being created and to reference it later. It should have meaning to the purpose of the report. If possible, avoid cryptic names. For example, “Cost Report” is a valid name, but it does not inform you of what costs you are reporting.

Since the report name is too short to be very descriptive, you may include a Description clause as part of the report definition. This enables you to associate a prompt, comment, or qualifying phrase with the report being defined.

For example, if you were defining a report named “Cost Report,” you might write an Add Report statement with a Description clause similar to one of the following. The information in each report might differ considerably.

```
add report "Cost Report"
description "Provides daily operating costs of the department";
add report "Cost Report"
description "Provides manufacturing costs for Widget A";
add report "Cost Report"
description "Provides monthly costs for supporting Widget B";
```

When specifying a value for the description, you can enter a string of any length. However, the longer the string, the more difficult it may be for the user to use.

Icon Clause

The Icon clause of the Add Report statement associates a special image with a report. Choose an icon that has meaning to the user. Icons can visually help a user locate the report s/he needs by clearly identifying the report function. The default icon is used when in view-by-icon mode. Any special icon you assign is used when in view-by-image mode. When assigning a unique icon, you must use a GIF image file. Refer to [Icon Clause](#) in Chapter 1 for a complete description of the Icon clause.

GIF filenames should not include the @ sign, as that is used internally by eMatrix.

Size Clause

The Size clause of the Add Report statement defines the page dimensions of the report. This is commonly equal to standard page sizes such as 8½ by 11 inches or 8½ by 14 inches. However, you are not restricted to these sizes.

A page can be any size that your printer can handle. A page is a logical unit that you define. Once defined, eMatrix will use that definition to determine where to place the header, footer, and margins. But, eMatrix must know the page size to determine when one page ends and another begins.

To define a page size, you need two numeric values. One represents the width (COL_SIZE) and one represents the length (ROW_SIZE). Both of these values must be provided and entered according to the following syntax:

```
size ROW_SIZE COL_SIZE
```

If you wanted the dimensions of a standard page, you would write one of the following clauses. The clause you use depends on the units you specified in the Units clause.

size 80 66	Measured in picas
size 612 792	Measured in points
size 8.5 11	Measured in inches

When specifying page dimensions, be sure the page break occurs in the correct place. For example, depending on your printer, a page may have 60 or 66 lines. For example, if your printer automatically inserts a footer and header value of 3 lines each, you should use a value of 60. Otherwise, you would end with a page of 72 lines (66 plus the footer and header) rather than 66. If you are unsure of your printer settings, try several test values to see what works best. Or, consult the printer user manual.

Header Clause

The Header clause of the Add Report statement places a border at the top of the page. It specifies the number of lines, points, or inches that should be measured down from the top of the page. While inserting a Header clause defines an upper border, it does not prevent you from placing information within that border.

Header clauses are often used in conjunction with display rules and page titles. When you define a header, you are defining a place where a rule line can be placed. You may want to place title information above the rule line with the values below it.

The following statement creates a report named “Material Properties.” Measured in picas, the report is 80 characters wide and 60 lines long. It has a header that extends six lines

down from the top of the page which is indicated by a dividing line (with the *Displayrule* clause described in *Displayrule Clause* below).

```
add report "Material Properties"
units picas
size 80 60
header 6
displayrule true;
```

When you define a large header value, you can ensure that a value is not printed directly on a page break. It is a good practice to allow some space near the expected page break. This allows for printer paper that is in a less than perfect position. Also, the border sets off each page and makes it easier to identify and read.

Footer Clause

The Footer clause of the Add Report statement is similar to the Header clause. It places a border at the bottom of the page by specifying the number of lines, points, or inches that should be measured up from the bottom of the page. While inserting a Footer clause defines a lower border, it does not prevent you from placing information within that border.

Footer clauses are often used in conjunction with display rules and page footnotes (such as page numbers or titles). When you define a footer, you are defining where a rule line can be placed. You may want to place your footnote information below that rule line. This information might consist of summary values, orientation material (such as a page number and title), or special information (such as warning flags).

A Footer clause can be used with or without a Header clause. When used alone, define a footer large enough to ensure that your values will not be printed directly on a page break.

For example, the following statement starts the report at the first line (a header is not defined). Then a border (defined by the footer) is placed six lines up from the bottom to allow for the page break.

```
add report "Material Properties"
units picas
size 80 60
footer 6
displayrule true;
```

But what if the printer paper is slightly below the desired starting point? In that case, you might lose the first line or two of report output. For that reason, it would be better to define the report as:

```
add report "Material Properties"
units picas
size 80 60
header 3
footer 3
displayrule true;
```

With this definition, the page will hold the same number of lines. However, the starting point is three lines down from the top to ensure that no field values are lost.

Margins Clause

The Margins clause of the Add Report statement specifies a left and right border on each page:

```
margins LEFT_MARGIN RIGHT_MARGIN
```

LEFT_MARGIN is the number of units eMatrix should move from the left page edge. Always specify this value first.

RIGHT_MARGIN is the number of units eMatrix should move from the right page edge.

For example, assume you want to include margins in a “Material Properties” report definition:

```
add report "Material Properties"  
units picas  
size 80 60  
header 3  
footer 3  
margins 5 10;
```

In this example, the left margin is set as five characters in from the left page edge. The right margin is set as ten characters in from the right page edge. Since the page size is set at 80 characters, this means that the center working area is equal to 65 characters.

When including a Margins clause in an Add Report statement, you must always specify two values even if you only want one margin. This enables eMatrix to determine which value is the left margin offset and which is the right. For example, to define only a right margin, you can use this Margins clause:

```
margins 0 5
```

A 0 value for the margin means that the first character is printed at the edge (coordinate 1).

The left margin is defined as the left page edge (0). The right margin is then defined as five characters in from the right page edge. This gives you a new working area of 75 characters in width.

Note that the margin values are relative to the page size. This means that you can apply borders to nonstandard page sizes. It also means that you can have a right or left border that is invalid or leaves no room for values. For example, assume you want to include a Margins clause within the “Label List” report definition:

```
add report "Label List"  
units picas  
size 40 12  
header 1  
footer 1  
displayrule true  
margins 5 5;
```

With the inclusion of the Margins clause, you have a working area that is 30 characters wide. The left margin is at five characters from the left edge and the right margin is at five characters from the right edge. Since the right edge is at 40 characters, this definition is equivalent to saying that the right margin is at 35 characters.

Displayrule Clause

The Displayrule clause displays a dividing line at the header and footer locations. This clause has two forms:

```
displayrule true  
Or:  
displayrule false
```

When the Displayrule clause is set to FALSE, a dividing line is not displayed. This is the default value. When the Displayrule clause is set to TRUE, eMatrix will print a line at the positions specified in the Header and Footer clauses. If Header and Footer clauses are not included in the definition, you will get an error.

Dividing lines can make a report easier to read by setting off important information (such as title or summary information). Since you can define a page to be of any size within the limitations of your printer, you can use the Displayrule clause to denote the boundaries of page information. For example, you may be printing labels that are eight lines in length. You may want to draw a dividing line between each label. To do so, you could write a statement such as:

```
add report "Label List"  
units picas  
size 40 12  
header 1  
footer 1  
displayrule true;
```

This report definition defines a page of only 12 lines in length and 40 characters wide. Since the header begins at one line down from the top and the footer begins one line up from the bottom, you can have a blank line before and after the eight lines of label information. The boundaries of each label will be marked by the two rule lines. If you only wanted one rule line to divide each label, you could use only a Header or Footer clause in the definition.

Field Clause

The Field clause specifies the values to be printed and their general placement on the page:

```
field FIELD_TYPE FIELD_DEF {FIELD_DEF}
```

FIELD_TYPE identifies the general function of the field value being defined. All Field clauses must include a Field Type subclause which must be given before any defining subclauses. When specified, the field type identifies the kind of value that will be printed:

Field Type	Meaning
string	A printable string of characters
expression	A database access expression that returns the desired database item
date	The current date taken from the system clock
calculated	A value that must be calculated from multiple field values

Each field type is defined in its own Field Type subclause, as discussed in the sections that follow.

`FIELD_DEF` (field definition) is a subclause that provides additional information about the value to be printed. Four different subclauses can define a field:

- Output subclause
- Geometry subclause
- Label subclause
- Filter subclause

These subclauses define information such as where the values should be placed on the page, how often the field values should be printed, and test criteria to ensure that you have the correct values. Each of these four subclauses and the arguments they use are discussed in the sections that follow.

Field Type Subclause

The Field Type Subclause consists of four different subclauses. Each identifies the type of field values that will be printed in the report. The `FIELD_TYPE` subclause must take one of these forms:

string STRING_VALUE
expression QUERY_WHERE_EXPRESSION
date
calculated sum NUMERIC_VAL_EXPR [resetperpage true false]
calculated average NUMERIC_VAL_EXPR [resetperpage true false]
calculated count QUERY_WHERE_EXPR [resetperpage true false]

String

The first form specifies that the field value is a text string or label. For example, a report title or footnote would use the String field type.

Expression

The second form specifies that the field value is the result of an expression. This expression is constructed according to the syntax described for the Where clause as described in [Query Overview](#) in Chapter 38.

Unlike a `QUERY_EXPR` expression that must produce a TRUE or FALSE value, the `QUERY_WHERE_EXPRESSION` can produce any type of value. This means that you can use the name of a field that contains a non-Boolean value in the field type definition. For example, each of the following are valid Expression subclauses that define a field type value:

expression DESCRIPTION
expression "All tests are negative"
expression '"attribute[Product Cost]" <= "attribute[Maximum Cost]"'
expression 'Blood_Test_Positive or EKG_Positive'

In the first example, the Description field value (a character string value) is used. In the second example, the value of the field named “All tests are negative” (a Boolean value) is printed. In the third and fourth examples, the values of the relational and Boolean expressions are used for the report output.

Valid expressions include attributes, descriptions, and other selectable items that can be described in a manner similar to “where” clauses in an MQL query. Refer to Appendix A, *Select Expressions*, in the *eMatrix Navigator Guide* for a list of selectables.

For example, you might want to create a form for Assembly types called Components Required to list information about components related to a selected assembly. You might want to include the name, type, revision, ImageIcon, and description of the selected Assembly object. Then, might want to include the name, description, revision, and quantity of each Component object that is related to the selected Assembly object.

When retrieving information from related objects, the number of values returned is the number of objects connected by the specified relationship to the selected object. When creating a form, be sure the field size is large enough to handle multiple entries.

For more information on writing query expressions, see [Query Overview](#) in Chapter 38. For more information on locating and specifying field names, refer to [Business Objects](#) in Chapter 34.

Date

The third form of the Field Type subclause specifies that the value is the date obtained from the system clock. When this field type is used, eMatrix prints the date using the format MM/DD/YY. This field type is useful to include the current date within your report.

Calculated

The remaining forms of the Field Type subclause—the Calculated subclauses—specify that the field value is calculated from other multiple field values. In the Calculated Sum/Average subclause, you can calculate either the total of the field values or the average value.

```
calculated sum NUMERIC_VAL_EXPR [resetperpage true|false]  
Or  
calculated average NUMERIC_VAL_EXPR [resetperpage true|false]
```

NUMERIC_VAL_EXPR is an expression that produces a numeric value. This expression follows the syntax specified for the Where clause as described in [Where Clause](#) in Chapter 38.

NUMERIC_VAL_EXPR can be the name of a field that contains a numeric value, an actual numeric value, or an arithmetic expression that results in a single numeric value.

These Calculated subclauses differ primarily in the choice of the second keyword:

- The keyword **Sum** means that the numeric values represented by NUMERIC_VAL_EXPR are to be added together into a single total.
- The keyword **Average** means that the numeric values represented by NUMERIC_VAL_EXPR should be totaled and divided by the number of values that were included.

The Calculated Count QUERY_WHERE_EXPR subclause uses a slightly different syntax:

```
calculated count QUERY_WHERE_EXPR [resetperpage true|false]
```

This specifies that the value should be a count of the number of values that meet the counting criteria. The keyword Count is used and the field value does not have to be numeric. You can use any valid query expression. If the expression returns a TRUE value, the count is incremented by one. If the value is FALSE, the count remains unchanged.

The expression also can provide information on how many values you have when the number is too large to count easily. And, it can flag errors or discrepancies that may be found.

In each of the three forms of Calculated subclause, you can use a resetperpage subclause to reset the total, average, or counter back to zero at the beginning of the next page if the resetperpage subclause is set to FALSE. A running total, average, or count will be compiled.

Field Definition (Output) Subclause

The Output subclause of the Add Report Field clause specifies how often the field value is to be printed in the report. The possible values are:

perline	The field value appears multiple times on a report page. The number of times is controlled by the size of the report page and the field values being printed.
pervalue	The field value appears on a report each time the report evaluation encounters valid data. Valid data must pass any evaluation requirements as well as any specified field filter expression. The report will continue to output the field values until the end of the page is reached. Then, the data reporting will continue on a new page.
perpage	The field value appears once per report page.
firstpage	The field value appears only on the first report page.
lastpage	The field value appears only on the last report page.

Field Definition (Geometry) Subclause

The Geometry subclause of the Add Report Field clause specifies the field size and where it is to be located on the report page:

```
geometry start START_POINT size ROW_SIZE COL_SIZE
```

START_POINT identifies the X and Y coordinates of the field's starting point. This is where the first character of the field value is printed.

ROW_SIZE specifies the horizontal size of the field.

COL_SIZE specifies the vertical size of the field.

The field's starting point can be specified in one of two ways. The first is to give the absolute X and Y coordinates. The second is to give the X and Y coordinates relative to the report's header and left margin.

The geometry size must be at least 1,1. Absolute coordinates begin with 1,1 and are measured from the upper left corner of the page.

Absolute coordinates use the syntax:

@X_START_VALUE @Y_START_VALUE

@ indicates that the coordinates are absolute.

X_START_VALUE specifies the distance across.

Y_START_VALUE specifies the distance down.

For example, you could write the following field descriptions to place a title at the top of the report and the current date:

field string "Daily Customer Report For: " output firstpage geometry start @10 @5 size 28 1 field date output firstpage geometry start @38 @5 size 9 1

The first description will start printing the title string “Daily Customer Report For:” in the upper left corner of the page. Its coordinates, given in picas, indicate ten characters over and five lines down from the uppermost left corner of the page. Following the title string is the current date. It is started on the same line 28 characters after the start of the title string. This starting location allows for the number of characters needed to print the title string. If the date location were less than 38, part of the title string would be overwritten. Take care to ensure that the field sizes do not conflict with field locations.

Relative coordinates can begin with 0,0 and are specified in the same general manner as absolute coordinates. Remember that, with relative coordinates, the values are measured from the upper left corner of the header and left margin intersection.

X_START_VALUE Y_START_VALUE

X_START_VALUE specifies the distance across.

Y_START_VALUE specifies the distance down.

For example, assume you have a report with a header of 6 and a left margin of 11. To place the same title and date as given in the previous example, you would write the following field definitions:

field string "Daily Customer Report For: " output firstpage geometry start 1 1 size 28 1 field date output firstpage geometry start 28 0 size 9 1
--

While the starting point is given as (1,1), this actually translates to an absolute starting point of (12,7). That is because the starting point for all relative coordinates is the bottom of the header (the 7th line) and the end of the left margin (12th character). When specifying the relative coordinates, you will always have to add the header and left margin values to obtain the absolute coordinates. Therefore a relative position of (28,0) translates into an absolute position of (39,6) with header and margin values of 6 and 11.

Note that there is no absolute starting point of 0,0.

When specifying the starting point, you can use any combination of relative or absolute values. Absolute coordinates are useful when you want to print a title within the header, footer, or margin areas. You cannot do this using relative coordinates. For example, to print the previous sample title above the display line, you would have to use absolute values such as:

```
add report "Customer List"
units picas size 80 60
header 6 margins 11 6
displayrule true
field string "Daily Customer Report For: "
    output firstpage
    geometry start @22 @3 size 27 1
field date
    output firstpage
    geometry start @49 @3 size 9 1;
```

This statement will place the title and date in the center of the page on line 3. Since the header is set off with a display rule clause, this gives you two lines above and two lines below the title.

Centering the title and date on the page is done by using the starting point in conjunction with the Size subclause. The Size subclause specifies the length and width of the field. First determine the number of characters required to print the title and date (36 characters) and then determine the amount of space remaining ($80 - 36 = 44$). That amount is divided in half to determine the starting row for the first field (@22). If you add the field size to this starting value, you find the starting location for the second field (@49).

Like the Start clause of the Add Report statement, the size is given row first and column second. For example, to print to field string "Address:" the size would be nine pica characters long and use one line. Therefore, its size could be expressed as:

```
size 9 1
```

All geometry subclauses must include the field size. If the size value is larger than the field value, the field value is padded with blank spaces so that the field and size values are equivalent. If the size value is smaller than the field value, the field value is truncated on the right to fit into the field size. Multiple-line text output will wrap at word boundaries if the report field is defined with a row size greater than 1.

As an added precaution, you could add a Filter subclause (discussed in [Field Definition \(Filter\) Subclause](#) below) to check for excessively large numbers. If one is encountered, you could print a warning message rather than the truncated value.

When specifying the number of rows used by a field, you should consider the spacing you want to use in the report. eMatrix will look at the number of rows required to print each field value on a line as well as the maximum number when determining how many lines must appear within each output group. For example, a sample report evaluation uses this report definition:

```
report "Material Properties Report"
description "Lists the material types of selected components and their total weight"
size 80 60
header 5 footer 5 margins 0 5
displayrule true
field string "Material Properties Report"
    output firstpage
    geometry start @5 @2 size 28 1
field date
```

```

output firstpage
geometry start @11 @3 size 9 1
label title "Date:"
    output firstpage
    geometry start @5 @3 size 6 1
field expression NAME
    output perline
    geometry start 5 4 size 15 2
label title "Name"
    output perpage
    geometry start 5 2 size 4 1field expression DESCRIPTION
output perline
geometry start 25 4 size 15 4
label title "Description"
    output perpage
    geometry start 25 2 size 11 1
field expression "attribute[Material Type]"
    output perline
geometry start 45 4 size 15 2
    label title "Material Type"
        output perpage
        geometry start 45 2 size 14 1
field expression "attribute[Target Weight]" - "attribute[Actual Weight]"
    output perline
    geometry start 63 4 size 7 1
    label title "Target -Actual"
        output perpage
        geometry start 63 2 size 15 1
field calculated sum "attribute[Actual Weight]" resetperpage true
    output perpage
geometry start 63 @56 size 15 1
label title "Total Weight"
    output perpage
geometry start 49 @56 size 20 1;

```

In this report evaluation, there are four field values printed per line: Name, Description, Material Type, and Target Weight. Of these four values, one (Description) requires four lines to print its entire value. Therefore, there are four lines in each output line. For the remaining fields, the additional lines appear blank.

Since eMatrix uses the maximum row size to determine line spacing, you may want to assign additional rows to a field value to control spacing between output lines. For example, a field may only require one line for printing. However, since you want a blank line between output lines, you might assign a row size of two instead of one. If you want double spacing, you could assign a row size of three, and so on. Remember that spacing adds to report readability when it is not excessive.

Field Definition (Label) Subclause

The Label subclause of the Add Report Field clause associates a label with the field. This label identifies the contents of the field to the reader. While labels are not required in a field definition, they make a report more readable.

For example, assume you are printing a telephone list for personnel. In this list, you include each person's name, phone number, and emergency contact. Without labels, the report would have two names and one telephone number. What is the relationship of the first name to the second? Labels can clearly identify these differences.

Labels provide a title for the associated field only when the associated field values appear in the report. Since labels are fields, they follow the same general syntax:

```
label title LABEL_NAME  
output OUTPUT_FREQUENCY  
geometry start START_POINT size ROW_SIZE COL_SIZE
```

LABEL_NAME is the title name assigned to the field.

OUTPUT_FREQUENCY identifies how often the title is to be printed. This frequency can be one of these values: Perline, Pervalue, Perpage, Firstpage, and Lastpage.

START_POINT identifies the X and Y coordinates of the field's starting point—where the first character of the field value is printed.

ROW_SIZE specifies the horizontal size of the field.

COL_SIZE specifies the vertical size of the field.

For example, a report might have this report definition:

```
report "Personnel Phone List"  
description "List of phone numbers and emergency contacts."  
units picas, size 80 60  
header 5, footer 5 margins 5 5  
displayrule true  
field string "Office Phone List For: "  
    output firstpage  
    geometry start @24 @3 size 23 1  
field date  
    output firstpage  
    geometry start @47 @3 size 9 1  
field expression NAME  
    output perline  
    geometry start 0 4 size 20 2  
    label title "Employee Name"  
        output perpage  
        geometry start 0 2 size 13 1  
field expression "attribute[Phone Number]"  
    output perline  
    geometry start 35 4 size 14 1;  
    label title "Phone Number"  
        output perpage  
        geometry start 28 2 size 12 1  
field calculated count "attribute[Emergency Contact]"  
    output perpage  
    geometry start 50 4 size 20 1  
    label title "Emergency Contact "  
        output perpage  
        geometry start 50 2 size 18 1;
```

In this example, each field has a title to help identify it to the user. Since the title is only required to identify the information in each column, the output frequency for each label definition is Perpage.

Field Definition (Filter) Subclause

The Filter subclause of the Add Report Field clause tests the field values to see if they meet a set of search criteria. For example, you may want to print a value only if it meets a set of conditions. The Filter subclause allows you to specify those conditions:

```
filter QUERY WHERE _EXPRESSION
```

QUERY WHERE _EXPRESSION is an expression that yields a Boolean value. It must obey the syntax for QUERY_EXPRS as defined in *Query Overview* in Chapter 38. If the expression does not yield a TRUE / FALSE value or does not obey the syntax for query expressions, an error message is displayed.

For example, assume you want to track of the number of customers that are pending and how many have been fully processed. To do this, you might check an attribute named “Customer Status.” If the attribute is set to Pending, you want to increment one counter. If the attribute is set to Processed, you want to increment a second counter. This could be done with the following two field definitions:

```
field calculated count `“attribute[Customer Status]” = Pending’  
    output lastpage  
    geometry start 72 @56 size 2 1  
    label title “Number of Customers Pending: ”  
        output lastpage  
        geometry start 40 @56 size 30 1  
field calculated count `“attribute[Customer Status]” = Processed’  
    output lastpage  
    geometry start 72 @57 size 2 1  
filter “attribute[Balance]” = 100  
    label title “Number of Customers With Balances Greater Than 100: ”  
        output lastpage  
        geometry start 40 @57 size 31 1
```

Both fields use the Customer Status field of a business object. However, when that field equals Processed and the filter’s query expression (`"attribute[Balance]" = 100') yields a TRUE value, the count is increased by one. If the filter expression yields a FALSE value, the count remains unchanged. When all of the objects have been evaluated, the values of each counter are printed with the appropriate labels.

Filter subclauses can also be used with Expression field types. For example, you may want to print the cost of an item. To do this, you could define a field as:

```
field expression `“attribute[Item Cost]”’  
    output perline  
    geometry start 65 4 size 5 1
```

However, what happens if you have a value too large to fit in the field’s size? If the value is larger than the size, the value is truncated. Therefore, you might want to test for any value that is out of the range of the expected item costs. If it is out of range, you could simply not print the value. The blank value would then represent a flag for out of range values.

To test for out of range values, include a Filter subclause in the above field definition as:

```
field expression `“attribute[Item Cost]”’  
    output perline  
    geometry start 65 4 size 5 1  
    filter `“attribute[Item Cost]”’
```

This subclause tests the Item Cost to see if it is less than \$100.00. If the expression result is TRUE, the Item Cost value is printed. If the expression result is FALSE, nothing is printed.

Hidden Clause

You can specify that the new report is “hidden” so that it does not appear in the Report chooser in eMatrix. You may want to use the hidden option if, for example, an object is under development or if it is intended only for your personal use. Hidden objects are also accessible through MQL.

Property Clause

Integrators can assign ad hoc attributes, called Properties, to the report. Properties allow associations to exist between administrative definitions that aren’t already associated. The property information can include a name, an arbitrary string value, and a reference to another administration object. The property name is always required. The value string and object reference are both optional. The property name can be reused for different object references, that is, the name joined with the object reference must be unique for any object that has properties.

```
add report NAME  
    property NAME [to ADMINTYPE NAME] [value STRING];
```

For additional information on properties, see *Overview of Administration Properties* in Chapter 24.

Evaluating a Report

Once you have defined a report, the next step is to evaluate it. Evaluating a report involves feeding business objects to the report for processing. This is done with the Evaluate Report statement:

```
evaluate report NAME BO_SOURCE [output FILENAME];
```

NAME specifies the report definition you want to use in the evaluation.

BO_SOURCE specifies the source of the business objects to be processed within the report.

FILENAME specifies an external file to be used to store the results of the report evaluation.

When an Evaluate Report statement is entered, eMatrix processes each of the business objects associated with the BO_SOURCE. This source may be either a query or a set. If the source is a query, the syntax of the Evaluate Report statement appears as:

```
evaluate report NAME query QUERY_NAME [output FILENAME];
```

If the source is a set, the syntax appears as:

```
evaluate report NAME set SET_NAME [output FILENAME];
```

A query produces a collection of business objects that meet specified search criteria. This collection then can be stored in a set. In both cases, a collection of business objects share a set of common features or values as specified by the query's search criteria.

When evaluating a report, you will want to match the report definition to the query's search criteria. If the business objects you are providing for the report do not have the correct types of values, the report results will be worthless.

For example, evaluating a report named "Current Customer Report" with business objects containing information on the storing and ordering components will only produce errors. Therefore, you should check the current query criteria or the set contents before evaluating a report to ensure that you will be working with the correct collection of business objects.

Once you have an appropriate collection of one or more business objects, you can have them evaluated to produce a set of report results.

Copying and/or Modifying a Report

Copying (Cloning) a Report Definition

After a report is defined, you can clone the definition with the Copy Report statement. This statement lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy report SRC_NAME DST_NAME [MOD_ITEM] {MOD_ITEM};
```

SRC_NAME is the name of the report definition (source) to copied.

DST_NAME is the name of the new definition (destination).

MOD_ITEMS are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

Modifying a Report

After a report is defined, you can change the definition with the Modify Report statement. This statement lets you add or remove defining clauses and change the value of clause arguments:

```
modify report NAME [MOD_ITEM {MOD_ITEM}];
```

NAME is the name of the report you want to modify.

MOD_ITEM is the type of modification you want to make. Each is specified in a Modify Report clause, as listed in the following table. Note that you only need to specify fields to be modified.

Modify Report Clause	Specifies that...
units [picas points inches]	The current units of measurement is changed to the new units entered.
name NEW_NAME	The current report name is changed to the new name entered
description VALUE	The current description value, if any, is set to the value entered.
icon FILENAME	The image is changed to the new image in the file specified.
size ROW_SIZE, COL_SIZE	The current page size is set to the new values given.
header HEADER_SIZE	The current header, if any is set to the value entered.
footer FOOTER_SIZE	The current footer, if any is set to the value entered.
margins LEFT_MARGIN RIGHT_MARGIN	The left and/or right margins are changed to the new values entered.
displayrule false	The header and footer rule lines are not displayed.
displayrule true	The header and footer rule lines are displayed.

Modify Report Clause	Specifies that...
field delete FIELD_NUMBER	The field identified by the given field number is removed from the report. To obtain the field number for a specific field, use the Print Report statement. When the report definition is listed, note the number assigned to the field to delete.
field add FIELD_TYPE FIELD_DEF { ,FIELD_DEF }	A new field is defined (according to the field definition clauses) and placed at the end of the field list.
hidden	The hidden option is changed to specify that the object is hidden.
nothidden	The hidden option is changed to specify that the object is not hidden.
property NAME [to ADMINTYPE NAME] [value STRING]	The named property is modified.
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.

Each modification clause is related to the arguments that define the report. To change the value of one of the defining clauses or add a new one, use the Modify clause that corresponds to the desired change. For example, use the Size clause of the Modify Report statement to alter the values of the Size clause used in the Add Report statement. The only exception to this general rule involves modifying field definitions.

When modifying field definitions within an existing report, you have only two choices. You can either remove an existing field definition or you can add a new one. The Modify Report clause does not offer a way to alter the subclause values that make up a field definition. Therefore, if you are unhappy with a subclause value, you can only remove the entire field definition and replace it with one that has the desired changes in it.

New field definitions appear at the end of the report definition. While they are listed last, their placement in the report definition does not affect the placement or printing of the report values. That is controlled by the geography and size values within the field definitions themselves.

When modifying a report, you can make the changes from a script or while working interactively with MQL. When making these changes, you should do one of the following:

- Perform one or two changes at a time if you are working interactively. This avoids the possibility of one invalid clause invalidating the entire statement.
- Group the changes together in a single Modify Report statement if you are working from a script.

Deleting a Report

If a report is no longer required, you can delete it by using the Delete Report statements

```
delete report NAME;
```

NAME is the name of the report to be deleted.

When this statement is processed, eMatrix searches the list of defined reports. If the name is found, that report is deleted. If the name is not found, an error message is displayed. For example, to delete the report named “Income Tax Report,” enter the following:

```
delete report "Income Tax Report";
```

After this statement is processed, the report is deleted and you receive an MQL prompt for another statement.

23

Working With Forms

Overview of Forms	586
Defining a Form	587
Units Clause	587
Description Clause.....	588
Icon Clause.....	589
Rule Clause	589
Color Clause	589
Header Clause.....	589
Footer Clause	590
Margins Clause.....	590
Type Clause.....	591
Size Clause.....	591
Field Clause.....	591
Hidden Clause	595
Property Clause	595
Copying and/or Modifying a Form	597
Copying (Cloning) a Form Definition.....	597
Modifying a Form	597
Deleting a Form	599
Printing a Form	600

Overview of Forms

A *form* is a window in which information related to an object is displayed. The Business Administrator designs the form, determining the information to be presented as well as the layout. Forms can be created for specific object types, since the data contained in different types can vary greatly. In addition, one object type may have several forms associated with it, each displaying different collections of information. When a user attempts to display information about an object by using a form, eMatrix only offers those forms that are valid for the selected object.

Expressions can be used in the form's field definitions to navigate the selected object's connections and display information from the relationship (attributes) or from the objects found at their ends. Forms can be used as a means of inputting or editing the attributes of the selected object. However, attributes of related objects cannot be edited in a form.

A special type of form called a Web form can be created for use in custom applications. Each field has several parameters where you can define the contents of the field, link data, user access, and other settings.

You must be a Business Administrator to define a form, and have form administrative access.

Defining a Form

Use the Add Form statement to define a form:

```
add form NAME [web] [ADD_ITEM {ADD_ITEM}];
```

NAME is the name you assign to the form you are creating. The form name is limited to 127 characters. For additional information, refer to *Business Object Name* in Chapter 34.

web is used when creating a “Web form.” This distinguishes forms to be used in HTML/JSP applications from those used in eMatrix thick client and PowerWeb.

ADD_ITEM is an Add Form clause which provides additional information about the form. The Add Form clauses are:

[! not] web
units [picas points inches]
description VALUE
icon FILENAME
rule RULENAME
color [FOREGROUND] [on BACKGROUND]
header HEADER_SIZE
footer FOOTER_SIZE
margins LEFT_MARGIN RIGHT_MARGIN
type TYPE_NAME { ,TYPE_NAME }
size WIDTH HEIGHT
field FIELD_TYPE FIELD_DEF
[! not] hidden
property NAME [to ADMINTYPE NAME] [value STRING]

With the exception of the Units and Size clauses, all other Add Form clauses are optional. (Units will default to picas if you do not enter a value.) Field clauses specify the field values that should be printed in the form and where. Without at least one Field clause, your form will not have much value.

The sections that follow explain more about each Add Form clause.

Units Clause

The Units clause of the Add Form statement specifies the units of page measurement. There are three possible values: picas, points, or inches.

```
units picas  
Or  
units points  
Or  
units inches
```

Without a unit of measurement, eMatrix cannot interpret the values of any given header, footer, margin, or field size. Because picas are the default unit of measurement, eMatrix will automatically assume a picas value if you do not use a Units clause.

Picas are the most common units of page measurement in the computer industry. Picas use a fixed size for all characters. Determining the size of a field value is easy when using picas as the measurement unit. Simply determine the maximum number of characters that will be used to contain the largest field value. Use that value as your field size. For example, if the largest field value will be a six digit number, you need a field size of six picas. This is not true when using points.

Points are standard units used in the graphics and printing industry. A point is equal to 1/72 of an inch or 72 points to the inch. Points are commonly associated with fonts whose print size and spacing varies from character to character. Unless you are accustomed to working with points, measuring with points can be confusing and complicated. For example, the character “T” may not occupy the same amount of space as the characters “E” or “O.” To determine the maximum field size, you need to know the maximum number of characters that will be used and the maximum amount of space required to express the largest character. Multiply these two numbers to determine your field size value.

Inches are common English units of measurement. While you can use inches as your unit of measurement, be aware that field placement can be difficult to determine and specify. Each field is composed of character string values. How many inches does each character need or use? If the value is a four-digit number, how many inches wide must the field be to contain the value? How many of these fields can you fit across a form page? Considering the problems involved in answering these questions, you can see why picas are a favorite measuring unit.

Description Clause

The Description clause of the Add Form statement provides general information to you and the user about the function of the form. There may be subtle differences between forms; you can use the Description clause to point out the differences to the user.

There is no limit to the number of characters you can include in the description. However, keep in mind that the description is displayed when the mouse pointer stops over the form in a chooser. Although you are not required to provide a description, this information is helpful when a choice is requested.

You can distinguish your form in your selection of a form name. This consists of a character string value to identify the form being created and to reference it later. It should have meaning to the purpose of the form. If possible, avoid cryptic names. For example, “Specification” is a valid name, but it does not inform you of the specifications for which the form was designed.

Since the form name it is too short to be very descriptive, you may include a Description clause as part of the form definition. This enables you to associate a prompt, comment, or qualifying phrase with the form being defined.

For example, if you were defining a form named “Cost Specification”, you might write an Add Form statement with a Description clause similar to one of the following. The information in each form might differ considerably.

```
add form "Cost Specification"
description "Provides costs for developing new product features for
Widget A";
```

```
add form "Cost Specification"
description "Provides all cost specifications for Widget A";
add form "Cost Specification"
description "Provides monthly costs for supporting field testing of
Widget B";
```

When specifying a value for the description, you may enter a string of any length.

Icon Clause

The Icon clause of the Add Form statement associates a special image with a form. Choose an icon that has meaning to the user. Icons can visually help a user locate the form s/he needs by clearly identifying the form function. You can assign a special icon to the new form or use the default icon. The default icon is used when in view-by-icon mode. Any special icon you assign is used when in view-by-image mode. When assigning a unique icon, you must use a GIF image file. Refer to [Icon Clause](#) in Chapter 1 for a complete description of the Icon clause.

GIF filenames should not include the @ sign, as that is used internally by eMatrix.

Rule Clause

Rules are administrative objects that define specific privileges for various eMatrix users. The Rule clause enables you to specify an access rule to be used for the form.

```
add form NAME rule RULENAME;
```

Color Clause

The Color clause of the Add Form statement specifies color values used as the default foreground and background for the form.

```
add form color [ FOREGROUND | on BACKGROUND | ]
```

FOREGROUND is the name of the color for the foreground printed information (any vertical or horizontal lines of information).

BACKGROUND is the name of the color used as an overall background for the form. Note that the word *on* is required only if a background color is specified.

For a list of available colors, refer to:

- Windows— `\$MATRIXHOME\lib\winnt\rgb.txt`.
- UNIX— `/$MATRIXHOME/lib/ARCH/rgb.txt`.

`\$MATRIXHOME` is where the eMatrix application is installed and `ARCH` is the UNIX platform.

Header Clause

The Header clause of the Add Form statement places a border at the top of the page. It specifies the number of lines, points, or inches that should be measured down from the top of the page. While inserting a Header clause defines an upper border, it does not prevent you from placing information within that border. Header clauses are often used in conjunction with page titles. You may want to place title information within the header with the values below it.

The following statement creates a form named “Material Properties.” Measured in picas, the form is 80 characters wide and 60 lines long.

```
add form "Material Properties"  
units picas  
size 80 60  
header 6
```

Footer Clause

The Footer clause of the Add Form statement is similar to the Header clause. It places a border at the bottom of the page by specifying the number of lines, points, or inches that should be measured up from the bottom of the page. While inserting a Footer clause defines a lower border, it does not prevent you from placing information within that border.

Footer clauses are often used in conjunction with display rules and page footnotes (such as page numbers or titles). When you define a footer, you are defining where a rule line can be placed. You may want to place your footnote information below that rule line. This information might consist of summary values, orientation material (such as a page number and title), or special information (such as warning flags).

```
add form "Material Properties"  
units picas  
size 80 60  
footer 6
```

Margins Clause

The Margins clause of the Add Form statement specifies a left and right border on each page:

```
add form margins LEFT_MARGIN RIGHT_MARGIN
```

LEFT_MARGIN is the number of units eMatrix should move from the left page edge. Always specify this value first.

RIGHT_MARGIN is the number of units eMatrix should move from the right page edge.

For example, assume you want to include margins in a “Material Properties” form definition:

```
add form "Material Properties"  
units picas  
size 80 60  
header 3  
footer 3  
margins 5 10;
```

In this example, the left margin is set as five characters in from the left page edge. The right margin is set as ten characters in from the right page edge. Since the page size is set as 80 characters, this means that the center working area is equal to 65 characters.

When including a Margins clause in an Add Form statement, you must always specify two values even if you only want one margin. This enables eMatrix to determine which value is the left margin offset and which is the right. For example, to define only a right margin, use can use this Margins clause:

```
margins 0 5
```

The left margin is defined as the left page edge (0). The right margin is then defined as five characters in from the right page edge. This gives you a new working area of 75 characters in width.

```
add form "Label List"
units picas
size 40 12
header 1
footer 1
margins 5 5;
```

With the inclusion of the Margins clause, you have a working area that is 30 characters wide. The left margin is at five characters from the left edge and the right margin is at five characters from the right edge. Since the right edge is at 40 characters, this definition is equivalent to saying that the right edge is at 35 characters.

Type Clause

The Type clause of the Add Form statement lists business types that the form is associated with. When a business object is highlighted in eMatrix and the Form option is selected, any forms associated with that type of business object are presented.

Size Clause

The Size clause of the Add Form statement defines the page dimensions of the form. This is commonly equal to standard page sizes such as 8½ by 11 inches or 8½ by 14 inches. However, you are not restricted to these sizes.

A page is a logical unit that you define. Once defined, eMatrix will use that definition to determine where to place the header, footer, and margins. But, eMatrix must know the page size to determine when one page ends and another begins.

To define a page size, you need two numeric values. One represents the width and one represents the height. Both of these values must be provided and entered according to the following syntax:

```
add form size WIDTH HEIGHT
```

If you wanted the dimensions of a standard page, you would write one of the following clauses. The clause you use depends on the units you specified in the Units clause.

size 80 66	Measured in picas.
size 612 792	Measured in points.
size 8.5 11	Measured in inches.

Field Clause

The Field clause specifies the values to be printed and their general placement on the page:

```
add form field FIELD_TYPE FIELD_DEF
```

`FIELD_TYPE` identifies the general function of the field value being defined. All Field clauses must include a Field Type subclause which must be given before any defining subclauses. When specified, the field type identifies the kind of value that will be printed:

Field Type	Meaning
<code>label STRING_VALUE</code>	A printable string of characters used for headers, column headings, and separators. For example, a form title or footnote would use the Label field type.
<code>select QUERY_WHERE_EXPR</code>	<p>Any selectable business object value. This allows for information to be retrieved from related objects as well as from the object to which the form is attached. Enter an expression that should appear on the form. Refer also to the list of selectables in the <i>Select Expressions</i> appendix of the <i>eMatrix Navigator Guide</i>.</p> <p>The expression is constructed according to the syntax described in Where Clause in Chapter 38.</p> <p>Unlike a <code>QUERY_EXPR</code> expression that must produce a true or false value, the <code>QUERY_WHERE_EXPRESSION</code> can produce any type of value. This means that you can use the name of a field that contains a non-Boolean value in the field type definition. For example, each of the following are valid Expression subclauses that define a field type value:</p> <pre>expression DESCRIPTION expression attribute "All tests are negative" expression `attribute[Base Cost]` <= "attribute[Maximum Cost]" expression 'Blood_Test_Positive or EKG_Positive'</pre> <p>In the first example, the <code>Description</code> field value (a character string value) is used. In the second example, the value of the attribute "All tests are negative" (a Boolean value) is printed. In the third and fourth examples, the values of the relational and Boolean expressions are used for the form output. For more information on writing query expressions, see Query Overview in Chapter 38. For more information on locating and specifying field names, refer to Business Objects in Chapter 34.</p>
<code>select edit true false</code>	Permits the editing of the field (edit true).
<code>select multiline true false</code>	Permits the output of multiple lines for a form field by adding scroll bars to the displayed field (multiline true).
<code>graphic IMAGE_PATH</code>	An imported graphical image, such as a logo or scanned image. Enter the directory path for the graphic file. This is the same image no matter what object is selected.
<code>image</code>	The <code>ImageIcon</code> of the business object to which the form is attached
<code>icon</code>	The icon of the business object to which the form is attached.

`FIELD_DEF` (field definition) is a subclause that provides additional information about the value to be printed. These subclauses define information such as where the values

should be placed on the page, how often the field values should be printed, and test criteria to ensure that you have the correct values.

Field Definition	Meaning
color [FOREGROUND] [on BACKGROUND]	The color of the form foreground (printed information) and background.
font FONT_NAME	The name of a system font that a field displaying text will use.
autoheight [false true]	Displays the height value for variable sized display fields.
autowidth [false true]	Displays the width value for variable sized display fields.
resizeheight [false true]	Permits the resizing of the height value for variable sized display fields.
resizewidth [false true]	Permits the resizing of the width value for variable sized display fields
drawborder [false true]	Draws a border around the output field.
minsize MIN_WIDTH MIN_HEIGHT	<p>The minimum width and/or height of the field.</p> <p><i>The mechanism used for rendering fonts on the Web differs from the one that is used for eMatrix on the desktop. Therefore, forms that are intended for use in both environments need to be designed to accommodate these slight differences. Increasing the size of the field will fix the problem.</i></p>
start XSTART YSTART	The X and Y coordinates of the field's starting point. This is where the first character of the field value is printed. Refer also to the description of Field Starting Point below.
size WIDTH HEIGHT	The width and height size of the form.
scale PERCENTAGE_VALUE	The percentage to scale the form.
href HREF_VALUE	For use in Web forms only to provide link data to the JSP.
alt ALT_VALUE	For use in Web forms only to display alternate text until any image associated with the command is displayed and also as "mouse over text."
range RANGE_HELP_HREF_VALUE	For use in Web forms only to specify the JSP that gets a range of values and populates the field with the selected value. These values can be displayed in a popup window or a combo box.
update UPDATE_URL_VALUE	For use in Web forms only to specify the URL page that should be displayed after the field is updated.
order NUMBER	For use in Web forms only to re-order field items.

Field Definition	Meaning
add user USER_NAME all	For use in Web forms only to specify who will be allowed access to the field.
remove user USER_NAME all	For use in Web forms only to specify who will not be allowed access to the field.
add setting NAME VALUE	For use in Web forms only. Settings are general name/value pairs that can be added to a field as necessary. They can be used by JSP code, but not by hrefs on the Link tab. Also refer to <i>Using Macros and Expressions in Dynamic UI Components</i> in the <i>eMatrix Business Modeler Guide</i> for more details.
remove setting NAME VALUE	For use in Web forms only to remove settings.

Field Starting Point

The field's starting point can be specified in one of two ways. The first is to give the absolute X and Y coordinates. The second is to give the X and Y coordinates relative to the form's header and left margin.

Absolute coordinates begin with 1,1 and are measured from the upper left corner of the page. They use the syntax:

```
@X_START_VALUE @Y_START_VALUE
```

@ indicates that the coordinates are absolute.

X_START_VALUE specifies the distance across.

Y_START_VALUE specifies the distance down.

For example, you could write the following field description to place a title at the top of the form:

```
field label "Daily Customer Form For:"  
start @10 @5 size 28 1
```

This description will start printing the title string "Daily Customer Form For:" in the upper left corner of the page. Its coordinates, given in picas, indicate ten characters over and five lines down from the uppermost left corner of the page. Take care to ensure that field sizes do not conflict with other field locations.

Relative coordinates can begin with 0,0 and are specified in the same general manner as absolute coordinates:

```
X_START_VALUE Y_START_VALUE
```

X_START_VALUE specifies the distance across.

Y_START_VALUE specifies the distance down.

However, with relative coordinates, the values are measured from the upper left corner of the header and left margin intersection.

For example, assume you have a form with a header of 6 and a left margin of 11. To place the same title in the same place as in the previous example, you would write the following field definition:

```
field string "Daily Customer Form For:"  
    start 0 0 size 28 1
```

While the starting point is given as (0,0), this actually translates to an absolute starting point of (11,6). That is because the starting point for all relative coordinates is the bottom of the header (the 6th line) and the end of the left margin (11th character). When specifying the relative coordinates, you will always have to add the header and left margin values to obtain the absolute coordinates. Therefore a relative position of (28,0) translates into an absolute position of (39,6).

When specifying the starting point, you can use any combination of relative or absolute values. Absolute coordinates are useful when you want to print a title within the heading, footer, or margin areas. You cannot do this using relative coordinates.

Centering the title on the page is done by using the starting point in conjunction with the Size subclause. The Size subclause specifies the width and height of the field. First determine the number of characters required to print the title (36 characters) and then determine the amount of space remaining ($80 - 36 = 44$). That amount is divided in half to determine the starting row for the first field (@22). If you add the field size to this starting value, you find the starting location for the second field (@49).

Like the Start clause of the Add Form statement, the size is given width first and height second. For example, a value of “Address:” is nine pica characters long and uses one line. Therefore, its size could be expressed as:

```
size 9 1
```

All geometry subclauses must include the field size. If the size value is larger than the field value, the field value is padded with blank spaces so that the field and size values are equivalent. If the size value is smaller than the field value, the field value is truncated on the right to fit into the field size. Multiple-line text output will wrap at word boundaries if the form field contains more than one output line.

Hidden Clause

You can specify that the new form is “hidden” so that it does not appear in the Forms chooser in eMatrix. You may want to use the hidden option if, for example, an object is under development or if it is intended only for your personal use. Hidden objects are accessible through MQL.

Property Clause

Integrators can assign ad hoc attributes, called Properties, to the form. Properties allow associations to exist between administrative definitions that aren’t already associated. The property information can include a name, an arbitrary string value, and a reference to another administration object. The property name is always required. The value string and object reference are both optional. The property name can be reused for different object references, that is, the name joined with the object reference must be unique for any object that has properties.

```
add form NAME  
    property NAME [to ADMINTYPE NAME] [value STRING];
```

For additional information on properties, see *Overview of Administration Properties* in Chapter 24.

Copying and/or Modifying a Form

Copying (Cloning) a Form Definition

After a form is defined, you can clone the definition with the Copy Form statement. This statement lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy form FROM_NAME TO_NAME [MOD_ITEM {MOD_ITEM}];
```

FROM_NAME is the name of the form definition (source) to copied.

TO_NAME is the name of the new definition (destination).

MOD_ITEMS are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

Modifying a Form

After a form is defined, you can change the definition with the Modify Form statement. This statement lets you add or remove defining clauses and change the value of clause arguments:

```
modify form NAME [MOD_ITEM {MOD_ITEM}];
```

NAME is the name of the form you want to modify.

MOD_ITEM is the type of modification you want to make. Each is specified in a Modify Form clause, as listed in the following table. Note that you only need to specify fields to be modified.

Modify Form Clause	Specifies that ...
units [picas points inches]	The current units of measurement is changed to the new units entered.
description VALUE	The current description value, if any, is set to the value entered.
icon FILENAME	The image is changed to the new image in the file specified.
add rule NAME	The named rule is added.
remove rule NAME	The named rule is removed.
color [FOREGROUND] [on BACKGROUND]	The foreground and/or background colors are changed to the new values entered.
header HEADER_SIZE	The current header, if any, is set to the value entered.
footer FOOTER_SIZE	The current footer, if any, is set to the value entered
margins LEFT_MARGIN RIGHT_MARGIN	The left and/or right margins are changed to the new values entered.
type TYPE_NAME { ,TYPE_NAME }	The type(s) associated with the form is changed to the type(s) entered.

Modify Form Clause	Specifies that ...
type delete TYPE_NAME {,TYPE_NAME}	The type identified by the type name is removed from the form.
size WIDTH HEIGHT	The current page size is set to the new values given
field delete FIELD_NUMBER	The field identified by the given field number is removed from the form. To obtain the field number for a specific field, use the Print Form statement. When the form definition is listed, note the number assigned to the field to delete.
field modify FIELD_NUMBER FIELD_DEF FIELD_TYPE FIELD_DEF	A field is modified (according to the field definition clauses) and placed at the end of the field list
hidden	The hidden option is changed to specify that the object is hidden.
nothidden	The hidden option is changed to specify that the object is not hidden.
property NAME [to ADMINTYPE NAME] [value STRING]	The named property is modified.
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.

Each modification clause is related to the arguments that define the form. To change the value of one of the defining clauses or add a new one, use the Modify clause that corresponds to the desired change. For example, use the Size clause of the Modify Form statement to alter the values of the Size clause used in the Add Form statement. The only exception to this general rule involves modifying field definitions.

When modifying field definitions within an existing form, you have only two choices. You can either remove an existing field definition or you can add a new one. The Modify Form clause does not offer a way to alter the subclause values that make up a field definition. Therefore, if you are unhappy with a subclause value, you can only remove the entire field definition and replace it with one that has the desired changes in it.

New field definitions appear at the end of the form definition. While they are listed last, their placement in the form definition does not affect the placement of the form values. That is controlled by the geography and size values within the field definitions themselves.

Deleting a Form

If a form is no longer required, you may delete it by using the Delete Form statement:

```
delete form NAME;
```

NAME is the name of the form to be deleted.

When this statement is processed, eMatrix searches the list of defined forms. If the name is found, that form is deleted. If the name is not found, an error message is displayed. For example, to delete the form named "Income Tax Form," enter the following:

```
delete form "Income Tax Form";
```

After this statement is processed, the form is deleted and you receive an MQL prompt for another statement.

Printing a Form

Use the Print Form statement to print information about the attributes of a specific form, including the number and characteristics of each form field.

```
print form NAME [SELECT];
```

NAME is the name of the form to be printed.

SELECT specifies a subset of the list contents. For a list of all the selectable fields for forms, see the *Select Expressions* appendix in the *eMatrix Navigator Guide*.

When this statement is processed, eMatrix searches the list of defined forms. If the name is found, that form information is printed. If the name is not found, an error message is displayed. For example, to print details about the form named “TechTip,” enter the following:

```
print form "TechTip";
```

The following is sample output:

```
MQL<28>print form 'TechTip';
```

```
Form           TechTip
inactive

field# 1      label Notes:
  font          Times New Roman-bold-12
  autoheight   false
  autowidth    false
  drawborder   false
  hidden        false
  start         2 54
  size          13 2
  user          all

field# 2      select attribute[Notes]
  color         black on lemon chiffon
  font          Times New Roman-10
  autoheight   false
  autowidth    false
  drawborder   true
  multiline    true
  edit          true
  hidden        false
  start         23 54
  size          75 5
  user          all

field# 3      label Reason:
  font          Times New Roman-bold-12
  autoheight   false
  autowidth    false
  drawborder   false
  hidden        false
  start         2 44
  size          20 2
  user          all
```

```

field# 4      select "attribute[Reason]"
  color        on lemon chiffon
  font         Times New Roman-10
  autoheight   false
  autowidth    false
  drawborder   true
  multiline    true
  edit         true
  hidden       false
  start        22 44
  size         75 8
  user         all

field# 5      select description
  color        on lemon chiffon
  font         Terminal-10
  autoheight   false
  autowidth    false
  drawborder   true
  multiline    true
  edit         true
  hidden       false
  start        22 23
  size         75 19
  user         all

nothidden
created Fri Jun 22, 2001 8:16:45 PM EDT
modified Fri Jun 22, 2001 8:16:45 PM EDT nothidden
created Wed Oct 31, 2001 2:57:09 PM EST
modified Wed Feb 20, 2002 2:47:56 PM EST

```

Since forms have additional uses in support of dynamic UI modeling, the MQL print command suppresses the output of data that is not used. For example, if you print a form that is defined as a system object used for Web applications, the following selects will not be printed:

`size, minsize, scale, font, minwidth, minheight, absolutex, absolutey, xlocation, ylocation, width, and height.`

Conversely, when printing non-Web forms, parameters used only for Web forms are suppressed from the output:

`href, alt, range, update, and settings`

24

Working With Administration Properties

Overview of Administration Properties	604
Defining a Property.....	605
Adding Properties to Administrative Definitions.....	606
Adding Properties to User Workspace Items.....	606
Modifying Properties	606
Deleting Properties	606
Listing Properties	606
Selecting Properties.....	607

Overview of Administration Properties

Ad hoc attributes, called Properties, can be assigned to an administrative object by users with Business Administration privileges. Properties allow links to exist between administrative definitions that aren't already associated. There are two kinds of properties:

- *User properties*, which can be created by users to suit their needs. They can apply to all administrative objects, including workspace objects.
- *System properties*, which come with eMatrix. These properties are used internally to implement certain kinds of administrative objects. For example, toolsets point to their programs via a property; views point to their components in the same way.

Properties may be useful for developers who are integrating eMatrix to other application programs. However, the typical Business Administrator may never have the need to use properties.

Properties can be created, modified, displayed, and deleted only through MQL. However, MQL can be embedded in programs, where clauses and select clauses, making properties available to a broader audience.

Defining a Property

Properties can be created and attached to an object at the same time using the `add property` command. A property must have a name and be “on” an object. It can, optionally, define a link to another administrative object using the “to” clause. This command, therefore, takes two forms, with and without the “to” clause.

```
add property NAME on ADMIN_TYPE ADMIN_NAME [system] [to  
ADMIN_TYPE ADMIN_NAME] [system] [value VALUE];
```

NAME is the name of the new property.

ADMIN_TYPE is the keyword for an administrative or workspace object:

association	group	policy	rule	toolset
attribute	index	process	server	type
command	inquiry	program	set	vault
cue	location	query	site	view
filter	menu	relationship	store	wizard
form	page	report	table	workflow
format	person	role	tip	

ADMIN_NAME is the name of the administrative object instance.

The `to ADMIN_TYPE ADMIN_NAME` is optional.

`system` is used only when adding properties on/to system tables.

`VALUE` is a string value of the property. The “value” clause is optional.

For creation and subsequent identification (modification, deletion, etc.) purposes, a property with a “to” clause is identified by the following arguments:

- the property NAME
- the “on” ADMIN_TYPE ADMIN_NAME
- the “to” ADMIN_TYPE ADMIN_NAME

While a property without a “to” clause is identified by only:

- the property NAME
- the “on” ADMIN_TYPE ADMIN_NAME

For example, Programs are associated to Formats inherently, since they make up part of the Format definition. But let’s say we want to add a Format property to a Program definition to indicate the type of environment required to execute it. We could add a property to a program as follows:

```
add property Format on Program Perlscript to Format Perl value yes;
```

A Format property could be added to other Programs as well. And other perl programs would be added “to” the Perl Format. However, the properties are unique in that the “on” object would differ.

Adding Properties to Administrative Definitions

A property can be added to an administrative object in three ways:

- It can be added using the property command, as shown above.
- In addition, a property can be added when an administrative object or workspace object is created, using the property clause with the add statement.
- A third way is within the modify statement for administrative/workspace objects.

```
add ADMIN_TYPE ADMIN_NAME add property NAME [to ADMIN_TYPE ADMIN_NAME] [value VALUE];  
  
modify ADMIN_TYPE ADMIN_NAME add property NAME [to ADMIN_TYPE ADMIN_NAME] [value  
VALUE];
```

For example, the following are equivalent to the command given above:

```
add program Perlscript add property Format to Format Perl value yes;  
  
modify program Perlscript add property Format to Format Perl value yes;
```

Adding Properties to User Workspace Items

Properties can be added to the logged on user's personal workspace objects. For example, the following adds the date property to the set MYSET:

```
add property Date to set Myset value 4/19/99;
```

Workspace objects include filters, cues, queries, tips, tables, sets, toolsets, and views.

Modifying Properties

The value of a property can be modified using either the `modify property` or `modify ADMIN` statements.

```
modify property NAME on ADMIN_TYPE ADMIN_NAME [to ADMIN_TYPE ADMIN_NAME] [value VALUE];  
  
modify ADMIN_TYPE ADMIN_NAME property NAME [to ADMIN_TYPE ADMIN_NAME] [value VALUE];
```

This command will create the property if it does not exist or will modify its value if it does. If other changes are required (for example, changing any ADMIN values) the property should be deleted and redefined.

Deleting Properties

Properties can be deleted with either of the following statements:

```
delete property NAME on ADMIN_TYPE ADMIN_NAME [to ADMIN_TYPE ADMIN_NAME];  
  
modify ADMIN_TYPE ADMIN_NAME remove property NAME [to ADMIN_TYPE ADMIN_NAME];
```

Listing Properties

Properties can be listed with the `list property` command, which takes the following forms:

```
list property [system] [on ADMIN_TYPE ADMIN_NAME];  
  
list property [system] [user person USER_NAME];
```

```
list property [system] [user all];
```

The **System** option is used to list the system-defined properties. Without it, only user-defined properties are listed.

The **user** option is for workspace properties. If **all** is indicated as the user, all properties on all Persons' workspace items will be displayed. **USER_NAME** is the name of the person.

The following should be noted:

- All properties on an administrative object or on a user's workspace objects can be listed.
- Currently only person users have workspace objects.
- The "on" and "user" clauses cannot be used together.
- The **all** keyword goes with the user clause—either a Person or **all** is specified.

To list all user properties

```
list property;
```

This will list all user properties on all non-workspace objects.

To list both system and user properties

```
list property system;
```

To list user properties on an administrative object

```
list property on ADMIN_TYPE ADMIN_NAME;
```

ADMIN_TYPE ADMIN_NAME here could be a workspace object (table, set, tip, etc.) of the current user, or a non-workspace object.

To list all properties of a Person's workspace objects

```
list property system user person USER_NAME;
```

Selecting Properties

The properties of administrative objects are selectable, using the following syntax:

```
print ADMIN_TYPE ADMIN_NAME select property;
```

The above will list all user properties associated with the specified administrative object, including their name, their "to" object, and their values. To further refine the list you can also select the following:

```
property.name  
property.value  
property.to
```

For example:

```
MQL<12>print program Perlscript select property;
program    Perlscript
    property = Format to Format Perl value 4
MQL<13>print program Perlscript select property.name;
program    Perlscript
    property[Format].name = Format
MQL<14>print program Perlscript select property.value;
program    Perlscript
    property[Format].value = 4
MQL<15>print program Perlscript select property.to;
program    Perlscript
    property[Format].to = Format Perl
```

25

Working With Aliases

Aliases Defined	610
Alias Properties.....	610
What Remains in Base Language	610
Enabling Aliases	611
Working with Language Aliases.....	612
Add Statement.....	612
Modify Statement.....	612
Delete Statement.....	612
Overriding the Language	613
Implementation Issues.....	613

Aliases Defined

eMatrix is in use around the world and is designed to support the use of multiple languages. Menus and messages can be translated and imported using the language import mechanism in the System Manager application. Double-byte characters are also supported for the wider Asian alphabets. But for global companies, with offices in several countries, there is a need to use the same database in several languages. Toward this end, aliases are provided to display and choose administrative object names in any number of languages, for desktop and eMatrix Navigator users. When displayed to the user, be it a human or a program, the chosen language is used. Likewise, any localized name, when fed into the system, will be mapped to its original name in the database.

Note that the term “language” is being used loosely here. There is no reason why two shops in the US can’t use different English terms to mean the same thing. One shop might use the term “widget” while the other shop uses the term “gadget.” There may even be a need to provide mappings in terminology between departments within the same shop. For example, a design department may use different terms than the manufacturing department, but still be talking about the same thing.

The chosen language can be temporarily overridden, such as during execution of a program, and then easily reset to the chosen language.

Aliases are not supported in applications that use the symbolic naming of the Application Exchange Framework, including MatrixOne’s Value Chain Portfolio applications. Java properties files are used for localizing schema in that environment, and aliases should not also be configured.

Alias Properties

An administrative object can have any number of alias properties, consisting of an identifier and an alias. That is, the name of the alias will identify the language, and its value of will be the administrative name in that language. You must be a Business Administrator to add aliases to any existing administrative object.

What Remains in Base Language

Aliases provide a powerful way to share data in a global economy. However, there are limits to what is included in this type of localization. The following stored text will be displayed in the base language:

- Business object attribute values
- Business object names and revisions
- Checked in files
- Object descriptions
- Program code
- MQL keywords
- eMatrix keywords used in `where` clauses when defining Queries, Cues, Tips, etc.

Since attribute ranges can be defined to be a program, attribute values could be programmatically translated at run time. Also, third party tools may be available for on-the-fly translations of descriptions and ingested files. However, keywords should NOT be translated in this way, as they would then not be recognized as such.

Enabling Aliases

The language preference applies to all words presented to the user through the eMatrix GUI: buttons, labels, messages, etc. This may or may not be the same as the alias preference. An alias may refer to the different names given to the same administrative objects according to the audience and not based on a traditional language. This is sometimes referred to as a “Schema” alias.

If using aliases for traditional localization only, you can enable the aliases by setting the MX_LANGUAGE_PREFERENCE property on a person, or in the .ini file. If both language and schema aliases are defined, you must also set the MX_ALIASNAME_PREFERENCE in the same way. These settings must match the defined alias names.

The alias for the current context user is checked in the following order:

- If MX_ALIASNAME_PREFERENCE is defined as a user property or is defined in the .ini file, and its value corresponds to a defined alias, it is used.
- Otherwise, if MX_LANGUAGE_PREFERENCE is defined as a user property or is defined in the .ini file, and its value corresponds to a defined alias, it is used.
- If neither of the above is found, and the user is logged in through the server, the language parameter from the client is checked to see if it corresponds to a defined alias. If so, it is used.
- As a last resort on desktop eMatrix, the deprecated variables MX_LANGUAGE_ALIASING and MX_LANGUAGE_CHOICE in the .ini file are used.

Refer to the *eMatrix Installation Guide* for more information.

Working with Language Aliases

Business Administrators can create any number of aliases for all administrative objects. This includes objects created in the System Manager application, as well as the Business Modeler Application. Aliases are added using the MQL application.

Add Statement

The add alias command is used to add a alias to any existing administrative object.

```
add alias NEWNAME to ADMIN_TYPE ADMIN_NAME [aliasname NAME];
```

NEWNAME is the new language word for the existing ADMIN_NAME.

ADMIN_TYPE ADMIN_NAME is the administrative object that is being translated.

NAME is the new language name that will be used as the search string. It must match the language choice variable in the initialization file in order to be active.

The aliasname clause is optional. If a NAME is not specified, the alias is added to the current language. If aliasing is not active or no language is chosen, the aliasname clause must be specified when adding an alias or an error will result.

For example:

```
add alias susie to person jones aliasname nicknames;
```

In the above example, the nicknames language would contain the alias “susie” for the person “jones.”

It is possible to give two different administrative objects of the same type the same alias name. This should be avoided, since doing so may result in unsatisfactory behavior.

Modify Statement

Aliases can be edited using the modify alias command:

```
modify alias NEWNAME on ADMIN_TYPE ADMIN_NAME [aliasname NAME];
```

For example, to change the alias susie to sue use:

```
modify alias sue on person jones aliasname nicknames;
```

When modifying aliases, notice that the current alias NAME need not be specified—the change is added to the specified language. If the aliasname clause is not included, the modification is made to the current language.

To modify the language name, you must add a new alias and then delete the old one.

Delete Statement

In order to delete aliases, they must be removed from each administrative object using the delete alias command:

```
delete alias NAME from ADMIN_TYPE ADMIN_NAME [aliasname NAME];
```

Overriding the Language

The current language can be controlled using the `push alias` and `pop alias` commands. This allows a program to temporarily suspend aliasing or redefine the language, and then restore the state of aliasing, without even having to know what it was.

```
push alias [NAME];
```

Pass a NAME parameter to redefine the current language. To turn off aliasing, simply push an alias with no name (that is, do not supply a language name).

To reset aliasing back to its original state, use the `pop` command:

```
pop alias;
```

For example, if aliasing is in use and the Business Administrator needs to set up another language, the program that adds the aliases for the new language should first `push` the current language, then add all the required new aliases, and finally `pop` the language to reset the original aliases.

Implementation Issues

Even when aliasing is active, the internal, native name of an administrative object can be used. However, confusion may arise if an alias for one object maps to the internal name of another. The system would find the alias first, and so that is what it would use. However, because of the possibility of unexpected results, this scenario should be avoided.

System performance will be affected when aliasing is turned on as a result of the internal search required for the mapping phase. As for storage issues related to alias properties, properties are extremely lightweight and this shouldn't be a concern.

26

Working With Pages

Overview.....	616
Defining a Page Object.....	617
Content Clause	617
Description Clause.....	618
File Clause	618
Icon Clause.....	618
Mime Clause.....	619
Hidden Clause	619
Property Clause	619
Copying (Cloning) a Page Definition	620
Modifying a Page Definition.....	621
Deleting a Page.....	622
Supporting Alternate Languages and Display Formats.....	623

Overview

A *page* is a type of eMatrix administrative object that is used to create and manage properties for Java applications. eMatrix first looks for a property file using the classpath, but if the file is not found, it looks for a page object of the same name. In a distributed environment, such as a RMI gateway configuration, this allows you to centralize all property files and propagate updates immediately.

In environments that use both desktop eMatrix and VCP applications, when emxSystem.properties and emxFrameworkStringResource.properties (including translated versions of the later) are stored in the database, certain errors can be avoided.

You can use MQL commands to create, delete, copy, modify, print, and list to manage page objects.

Page objects currently are not supported in a J2EE environment.

Defining a Page Object

As a Business Administrator, you can create new page objects that can be used in Web page design. A page object requires code that defines a page or a part of a page. There are two ways to specify the code for a page:

- Enter the code as value for the `content` command.
- Write the code in another editor and save the file, then include the name of the file in the `file` command.

Use the `Add Page` command to define a new page:

```
add page NAME [ADD_ITEM {ADD_ITEM}];
```

NAME is the name you assign to the page. Page names must be unique. The name you choose is the name that will be referenced to include this page within a Web page. The page name is limited to 127 characters.

ADD_ITEM is an `Add Page` clause that provides additional information about the page you are defining. The `Add Page` clauses are:

content VALUE
description VALUE
file FILENAME
icon FILENAME
mime VALUE
[! not]hidden
property NAME [to ADMINTYPE NAME] [value STRING]

All clauses are optional for defining a page, since the page can later be modified. But to be used in a Web page, you must include either content or a file. Each clause and the arguments they use are discussed in the sections that follow.

Content Clause

The Content clause of the `Add Page` command is used to add code that defines the Web page. Content can consist of embedded tags and text.

```
content VALUE;
```

VALUE can be any combination of code and text that is displayable in a Web browser, including HTML, XHTML, XML, JavaScript, eMatrix tags, CSS, etc. If the code contains embedded double quotes, use single quotes to define the start and end of the content.

For example, you can define the following page, which might be included as a footer on every page within an application:

```
add page IncludeFooter
  content '<a href="http://www.XYZCorp.com">XYZ Corporation</a><br>
  Voice: (555) 123-4567<br>
  Fax: (555) 123-4568<br>
  <a href="mailto:support@XYZCorp.com">support@XYZCorp.com</a><br>
  <a href="mailto:sales@XYZCorp.com">sales@XYZCorp.com</a><br>'
```

As an alternative, you can write the code for the page in a separate file and use the [File Clause](#) to include the page content in the page definition.

Legal characters in XML are the tab, carriage return, line feed, and the legal graphic characters of Unicode, that is, #x9, #xA, #xD, and #x20 and above (HEX). Therefore, other characters, such as those created with the ESC key, should not be used for ANY field in eMatrix, including business and administrative object names, description fields, program object code, or page object content.

Description Clause

The Description clause of the Add Page command provides general information for you and the user about the function, use, or content of the page. There may be subtle differences between pages; the description clause points out the differences to the user.

The description can consist of a prompt, comment, or qualifying phrase. There is no limit to the number of characters you can include in the description.

For example, if you were defining a page named “Cost Specification”, you might write an Add Page statement with a description clause similar to one of the following. The information in each page might differ considerably.

```
add page "Cost Specification"
  description "Costs for developing new product features for Widget A";
add page "Cost Specification"
  description "Cost specifications for Widget A";
add page "Cost Specification"
  description "Monthly costs for supporting field testing of Widget B";
```

File Clause

The File clause of the Add Page command specifies a file that contains the code that constitutes the page. This is provided as an alternative to the [Content Clause](#). With the File clause, you can type and save the code in any editor of your choice.

```
file FILENAME;
```

For example, you might create a file that contains the frameset that is used to display your Web pages. You could create a page object named InitialFrame to store this data.

```
add page InitialFrame
  file InitialFrame.jsp;
```

Icon Clause

You can assign a special icon to the new page. Icons help users locate and recognize items. When assigning an icon, you must select a GIF format file.

The icon assigned to a page is also considered the ImageIcon of the page. When an object is viewed as either an icon or ImageIcon, the GIF file associated with it will be displayed.

Mime Clause

The **Mime** clause of the Add Page command associates a MIME (Multi-Purpose Internet Mail Extension) type with a page. It defines the content type of the file named in the [File Clause](#).

```
mime VALUE;
```

VALUE is the content type of the file. The format of value is a type and subtype separated by a slash. For example, text/plain or text/jsp.

The major MIME types are application, audio, image, text, and video. There are a variety of formats that use the application type. For example, application/x-pdf refers to Adobe Acrobat Portable Document Format files. For information on specific MIME types (which are more appropriately called “media” types) refer the Internet Assigned Numbers Authority Website at <http://www.isi.edu/in-notes/iana/assignments/media-types/>. The IANA is the repository for assigned IP addresses, domain names, protocol numbers, and has also become the registry for a number of Web-related resources including media types.

Hidden Clause

You can specify that the new page is “hidden” so that it does not appear in the chooser in eMatrix. Users who are aware of the hidden page’s existence can enter its name manually where appropriate. Hidden objects are accessible through MQL.

Property Clause

Integrators can assign ad hoc attributes, called properties, to the page. Properties allow associations to exist between administrative definitions that aren’t already associated. The property information can include a name, an arbitrary string value, and a reference to another administration object. The property name is always required. The value string and object reference are both optional. The property name can be reused for different object references, that is, the name joined with the object reference must be unique for any object that has properties.

```
add page NAME
    property NAME [to ADMINTYPE NAME] [value STRING];
```

For additional information on properties, see [Chapter 24, Working With Administration Properties](#).

Copying (Cloning) a Page Definition

After a page is defined, you can clone the definition with the `Copy Page` command. This command lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy page SRC_NAME DST_NAME [MOD_ITEM {MOD_ITEM}] ;
```

`SRC_NAME` is the name of the page definition (source) to copied.

`DST_NAME` is the name of the new definition (destination).

`MOD_ITEMS` are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

Modifying a Page Definition

Use the `Modify Page` command to change the definition of an existing page object. This command lets you add or remove defining clauses and change the value of clause arguments:

```
modify page NAME [MOD_ITEM {MOD_ITEM}];
```

`NAME` is the name of the page you want to modify.

`MOD_ITEM` is the type of modification you want to make. Each is specified in a `Modify Page` clause, as listed in the following table. Note that you need to specify only fields to be modified.

Modify Page Clause	Specifies that...
<code>name NEW_NAME</code>	The current page name changes to the new name entered.
<code>content VALUE</code>	The current page content is replaced with new content.
<code>description VALUE</code>	The current description value, if any, is changed to the value entered.
<code>icon FILENAME</code>	The image is changed to the new image in the file specified.
<code>file FILENAME</code>	The page file is changed to the new file specified.
<code>mime MIMETYPE</code>	The MIME type for the page is changed to the new value specified.
<code>hidden</code>	The hidden option is changed to specify that the object is hidden.
<code>nohidden</code>	The hidden option is changed to specify that the object is not hidden.
<code>property NAME [to ADMINTYPE NAME] [value STRING]</code>	The named property is modified.
<code>add property NAME [to ADMINTYPE NAME] [value STRING]</code>	The named property is added.
<code>remove property NAME [to ADMINTYPE NAME] [value STRING]</code>	The named property is removed.

As you can see, each modification clause is related to the clauses and arguments that define the page. For example, you would use the `Name` clause of the `Modify Page` command to change the name for the page file.

Deleting a Page

If a page is no longer required, you can delete it by using the `Delete Page` command:

```
delete page NAME;
```

NAME is the name of the page to be deleted.

When this command is processed, eMatrix searches the list of defined pages. If the name is found, that page is deleted. If the name is not found, an error message is displayed.

For example, to delete the `AddStateFrame` page, enter the following command:

```
delete page "AddStateFrame";
```

After this command is processed, the page is deleted and you receive an MQL prompt for another command.

Supporting Alternate Languages and Display Formats

For global database access, pages generally need to be provided in multiple languages. And with the wide use of cell phones and other hand-held devices in accessing Web pages, you may also need to support the page's display on a small LCD in wireless markup language (wml). When this is the case, you can use the following ADK call to open a page with a language and format argument, following the syntax:

```
open(BASE_PAGE_NAME, LANG, MIMETYPE)
```

For example:

```
open(login.jsp, fr, wml)
```

When evaluating this code, the page servlet first looks for the file named `login_fr_wml.jsp`. If this page is not found, it then attempts to find `login_wml.jsp`. As a last resort, it searches for the page `login.jsp`. Of course, you could call `login_fr_wml.jsp` directly, but the addition of arguments gives you much more flexibility when writing the code.

In this case, you would first create `login.jsp`. Next, if you wanted to support wml, you would then create the wireless version of the page and name it `login_wml.jsp`. Then for each language you want to support, you would translate the text portions of the page(s) and save as `login_LANG.jsp` and `login_LANG_wml.jsp`. For example, to support multiple languages you might have pages with the following names in the database:

```
login.jsp  
login_ch-tw.jsp  
login_ch-gb.jsp  
login_it.jsp
```

To then add support for wml for these languages, you might add the following pages:

```
login_wml.jsp  
login_ch-tw_wml.jsp  
login_ch-gb_wml.jsp  
login_it_wml.jsp
```

Note that the base page does not have to be in English. Also, the LANG argument could be more than two characters, such as en-us, en-uk, or ch-tw.

Working With Resources

Overview.....	626
Defining a Resource	627
Description Clause.....	627
File Clause.....	627
Mime Clause.....	628
Icon Clause.....	628
Hidden Clause	628
Property Clause	628
Copying (Cloning) a Resource Definition.....	629
Modifying a Resource Definition.....	630
Deleting a Resource	631
Supporting Alternate Languages and Display Formats.....	632

Overview

A *resource* is an eMatrix administrative object that stores binary files of any type and size. eMatrix applications use resources to display output to a standard Web browser or a small LCD device by providing components for Web pages. They are often .gif images, but they could be movie or video files or any resource that you use in a Web application, including:

- GIF
- JPEG
- MPEG
- AVI
- WAV
- JAR
- CAB

For example, you can include in the database a resource that represents the company corporate logo. In a Web application, the company corporate logo may be referred to many times.

The resource editor allows you to name the administrative definition, give it a description, and define a MIME (Multi-Purpose Internet Mail Extension) type that is used to ensure that the browsers know what kind of component this is. For example, the company corporate logo could be an *image/gif* MIME type, indicating that it is a .gif file that should be rendered in the browser.

You can use standard MQL commands such as create, delete, copy, modify, print, and list to manage resources. You can also create, modify, and delete resources using the eMatrix Business Modeler application. Specialized functions and embedded commands facilitate evaluation, translation, or formatting of eMatrix objects or output on HTML pages.

Resource objects currently are not supported in a J2EE environment.

Defining a Resource

Use the Add Resource command to define a new resource:

```
add resource NAME [ADD_ITEM {ADD_ITEM}];
```

NAME is the name you assign to the resource. The name you choose is the name that will be referenced to include this resource within a Web page. Resource names must be unique. The resource name is limited to 127 characters.

ADD_ITEM is an Add Resource clause that provides additional information about the resource you are defining. The Add Resource clauses are:

description VALUE
file FILENAME
mime VALUE
icon FILENAME
[! not]hidden
property NAME [to ADMINTYPE NAME] [value STRING]

All clauses are optional for defining a resource, since the resource can later be modified. But to be used in a Web page, at least the file and MIME type must be defined. Each clause and the arguments they use are discussed in the sections that follow.

Description Clause

The Description clause of the Add Resource command provides general information for you and the user about the function of the resource. There may be subtle differences between resources; the Description clause points out the differences to the user.

The description can consist of a prompt, comment, or qualifying phrase. There is no limit to the number of characters you can include in the description.

For example, if you were defining a resource named “Corporate Logo”, you might write an Add Resource statement with a Description clause similar to one of the following.

```
add resource "Corporate Logo"  
    description "Small size for use in page footer";  
  
add resource "Corporate Logo"  
    description "Large size logo for use in banners";
```

File Clause

The File clause of the Add Resource command defines the binary file that contains the resource. This is often a .gif or other image file, but they can be any resource that you use in a Web application.

```
file FILENAME;
```

For example, to define a resource for the company logo, you could use:

```
add resource Logo  
    file eMatrixlogo.gif;
```

Mime Clause

The mime clause of the Add Resource command associates a MIME (Multi-Purpose Internet Mail Extension) type with a resource. It defines the content type of the file.

```
mime VALUE;
```

Value is the content type of the file. The format of value is a type and subtype separated by a slash. For example, image/gif or video/mpeg.

The major MIME types are application, audio, image, text, and video. There are a variety of formats that use the application type. For example, application/x-pdf refers to Adobe Acrobat Portable Document Format files. For information on specific MIME types (which are more appropriately called “media” types) refer the Internet Assigned Numbers Authority Website at <http://www.ietf.org/rfc/rfc2046.txt>. The IANA is the repository for assigned IP addresses, domain names, protocol numbers, and has also become the registry for a number of Web-related resources including media types.

Icon Clause

You can assign a special icon to the new resource or use the default icon. The default icon is used when in view-by-icon mode. Any special icon you assign is used when in view-by-image mode.

GIF filenames should not include the @ sign, as that is used internally by eMatrix.

Hidden Clause

You can specify that the new resource is “hidden” so that it does not appear in the chooser in eMatrix. Users who are aware of the hidden resource’s existence can enter its name manually where appropriate. Hidden objects are accessible through MQL.

Property Clause

Integrators can assign ad hoc attributes, called properties, to the resource. Properties let you define associations between administrative objects. The property information can include a name, a string value, and a reference to another administration object. The property name is always required. The value string and object reference are both optional.

```
add resource NAME  
property NAME [to ADMINTYPE NAME] [value STRING];
```

For additional information on properties, see [Chapter 24, Working With Administration Properties](#).

Copying (Cloning) a Resource Definition

After a resource is defined, you can clone the definition with the Copy Resource command. This command lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy resource SRC_NAME DST_NAME [MOD_ITEM {MOD_ITEM}] ;
```

SRC_NAME is the name of the resource definition (source) to copied.

DST_NAME is the name of the new definition (destination).

MOD_ITEMS are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

Modifying a Resource Definition

Use the Modify Resource command to change the definition of an existing resource. This command lets you add or remove defining clauses and change the value of clause arguments:

```
modify resource NAME [MOD_ITEM {MOD_ITEM}];
```

NAME is the name of the resource you want to modify.

MOD_ITEM is the type of modification you want to make. Each is specified in a Modify Resource clause, as listed in the following table. Note that you need to specify only fields to be modified.

Modify Resource Clause	Specifies that...
name NEW_NAME	The current resource name changes to the new name entered.
description VALUE	The current description value, if any, is set to the value entered.
icon FILENAME	The image is changed to the new image in the file specified.
file FILENAME	The resource file is changed to the new file specified.
mime VALUE	The mime type for the resource is changed to the new value specified.
hidden	The hidden option is changed to specify that the object is hidden.
nothidden	The hidden option is changed to specify that the object is not hidden.
property NAME [to ADMINTYPE NAME] [value STRING]	The named property is modified.
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.

As you can see, each modification clause is related to the clauses and arguments that define the Resource.

Deleting a Resource

If a resource is no longer required, you can delete it using the Delete Resource command:

```
delete resource NAME ;
```

NAME is the name of the resource to be deleted.

When this command is processed, eMatrix searches the list of defined resources. If the name is found, that resource object is deleted. If the name is not found, an error message is displayed.

For example, to delete the Logo resource, enter the following command:

```
delete resource "Logo" ;
```

After this command is processed, the Logo resource object is deleted.

Supporting Alternate Languages and Display Formats

For global database access, resources generally need to be provided in multiple languages. And with the wide use of cell phones and other hand-held devices in accessing Web pages, you may also need to support the resource's display on a small LCD in wireless markup language (wml). When this is the case, you can use the following ADK call to open a resource with a language and format argument, following the syntax:

```
open( BASE_RESOURCE_NAME, LANG, MIMETYPE )
```

For example:

```
open(logo.gif, fr, wml)
```

When evaluating this code, the resource servlet first looks for the file named `logo_fr_wml.gif`. If the servlet does not find this resource object, it attempts to find `logo_wml.gif`. As a last resort, it searches for the resource `logo.gif`. Of course, you could call `login_fr_wml.gif` directly, but the addition of arguments gives you much more flexibility when writing the code.

In this case, you would first create `logo.gif`, and then create the wireless version of the resource (generally a much smaller version) and name it `logo_wml.gif`. Then, for each language that requires a different version of the resource (a translated textual image, a sound byte, or even a pure image that, due to cultural differences, requires a localized version) you would create the different resource files and reference them in a new resource object and save as `logo_LANG.gif` and `logo_LANG_wml.gif`. For example, to support multiple languages, you may have resources with the following names in the database:

```
logo.gif  
logo_ch-tw.gif  
logo_ch-gb.gif  
logo_it.gif
```

To add support for wml, you might add the following resource:

```
logo_wml.gif
```

In this case, all languages would use the same wml resource.

Note that the base resource does not have to be in English. Also, the LANG argument could be more than two characters, such as en-us, en-uk, or ch-tw.

28

Working With Commands

Commands Defined	634
Creating a Command.....	635
Description Clause.....	635
Icon Clause.....	636
Label Clause.....	636
Href Clause.....	636
Alt Clause	636
Code Clause	637
File Clause.....	637
User Clause.....	637
Setting Clause	637
Property Clause	638
Copying and/or Modifying a Command.....	639
Copying (Cloning) a Command Definition	639
Modifying a Command.....	639
Deleting a Command.....	641
Using Macros and Expressions in Configurable Components	642
Supported Macros and Selects.....	643

Commands Defined

eMatrix Business Administrators can create new command objects if they have the Menu administrative access. Commands can be used in any kind of menu in a JSP application. Commands may or may not contain code, but they always indicate how to generate another Web page. Commands are child objects of menus — commands are created first and then added to menu definitions, similar to the association between eMatrix types and attributes. Changes made in any definition are instantly available to the applications that use it.

Commands can be role-based, that is, only shown to particular users. For example, a number of commands may only be available when a person is logged in as a user defined as Administrator. When no users are specified in the command definitions, they are globally available to all users.

Creating a Command

To define a command from within MQL use the Add Command statement:

```
add command NAME [ADD_ITEM {ADD_ITEM}];
```

NAME is the name of the command you are defining. Command names cannot include asterisks. The name you choose is the name that will be referenced to include this command within a menu.

You cannot have both a command and a menu with the same name.

ADD_ITEM is an Add Command clause that provides additional information about the command. The Add Command clauses are:

description STRING_VALUE
icon IMAGE_PATH
label VALUE
href VALUE
alt VALUE
code VALUE
file FILENAME
user [NAME {,NAME} all]
setting NAME [STRING]
property NAME on ADMIN [to ADMIN] [value STRING]

Each clause and the arguments they use are discussed in the sections that follow.

Description Clause

The Description clause of the Add Command statement provides general information about the function of the command. Since there may be subtle differences between commands, you can use the description clause to point out the differences.

The description can consist of a prompt, comment, or qualifying phrase. There is no limit to the number of characters you can include in the description.

For example, if you were defining a command named “Cost Evaluator” you might write a Description clause similar to one of the following. Each command might differ considerably.

description "Figures cost based on wholesale prices"
description "Figures cost based on discounted prices"
description "Figures monthly costs for field testing of Widget B"

Icon Clause

You can assign a special icon to the new command. Icons help Business Administrators locate and recognize items. When assigning an icon, you must select a .gif format file. The icon assigned to a command is used only within Business Modeler and not in the Web page that contains the command.

The icon assigned to a command is also considered the ImageIcon of the command. When an object is viewed as either an icon or ImageIcon, the .gif file associated with it will be displayed.

Label Clause

The Label clause of the Add Command statement specifies the label to appear in the menu in which the command is assigned. For example, many desktop applications have a File menu with options labeled “Open” and “Save.”

Href Clause

The Href clause of the Add Command statement is used to provide link data to the href HTML commands on the Web page that references the command. This field is optional, but generally an href value is included.

The syntax is:

```
href VALUE;
```

VALUE is the link data.

For example:

```
href "emxForm.jsp?header=Basic Info";
```

Assigning an href to a link can create problems if a user clicks the same link twice when initiating such actions as changing an object’s state or submitting a form. The reason for this is that an href assigned to a link is considered a server request, even if it is a JavaScript command. Whenever the browser detects a server request, the browser stops processing the current request and sends the new request. Therefore, when a user first clicks on an href link, the request is processed, and typically, a JSP page starts executing. If, during this time, a user clicks the same link again, the first request is interrupted before completion and the new request is processed instead.

To avoid this scenario, you can set the href to “#” and use the onclick event instead. The generic code for this is:

```
<a href="#" onclick="submitForm()">
```

Alt Clause

The Alt clause of the Add Command statement is used to define text that is displayed until any image associated with the command is displayed and also as “mouse over text.”

The syntax is:

```
alt VALUE;
```

VALUE is the text that is displayed.

For example, you could use the following for a Tools command:

```
alt "Tools";
```

Code Clause

The Code clause of the Add Command statement is used to add JavaScript code to the command.

The syntax is:

```
code VALUE;
```

VALUE is the JavaScript code.

When commands are accessed from a JSP page, the href link is evaluated to bring up the next page. Commands only require code if the href link references JavaScript that is not provided on the JSP.

You can add the code in the Code clause or it can be written in an external editor. If you use an external editor, use the [File Clause](#) instead.

File Clause

The File clause of the Add Command statement is used to specify the file that contains the code for the command when the code is written in an external editor.

The syntax is:

```
file FILENAME;
```

FILENAME is the name of the file that contains the code for the command.

User Clause

The User clause of the Add Command statement is used to define the users allowed to see the command.

Any number of roles, groups, persons, and/or associations can be added to the command (role-based in this case includes all types of users and is not limited to only eMatrix roles).

The syntax is:

```
user NAME { ,NAME };
```

NAME is the name of the user(s) who have access to see the command.

Or

```
user all;
```

If the user clause is not included, all users are given access to the command.

Setting Clause

The Setting clause of the Add Command statement is used to provide any name/value pairs that the menu may need. They can be used by JSP code, but not by hrefs on the Link tab. Refer to *Parameters and Settings for Commands* in the *Application Exchange Framework Guide* for appropriate settings for the type of menu you are creating. Also refer to [Using Macros and Expressions in Configurable Components](#) for more details.

The syntax is:

```
setting NAME [ STRING ];
```

For example, an image setting with the image name can be specified to display when the menu is used in a toolbar:

```
setting Image iconSmallMechanicalPart.gif;
```

Property Clause

Integrators can assign ad hoc attributes, called Properties, to the command. Properties allow associations to exist between administrative definitions that aren't already associated. The property information can include a name, an arbitrary string value, and a reference to another administration object. The property name is always required. The value string and object reference are both optional. The property name can be reused for different object references, that is, the name joined with the object reference must be unique for any object that has properties.

```
add command NAME property NAME [to ADMINTYPE NAME] [value STRING];
```

For additional information on properties, see *Overview of Administration Properties* in Chapter 24.

Copying and/or Modifying a Command

Copying (Cloning) a Command Definition

After a command is defined, you can clone the definition with the Copy Command statement. Cloning a Command definition requires Business Administrator privileges, except that you can copy a Command definition to your own context from a group, role or association in which you are defined.

This statement lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy command SRC_NAME DST_NAME [MOD_ITEM {MOD_ITEM}];
```

SRC_NAME is the name of the command definition (source) to be copied.

DST_NAME is the name of the new definition (destination).

MOD_ITEMS are modifications that you can make to the new definition. Refer to the command below for a complete list of possible modifications.

Modifying a Command

The List Command statement is used to display a list of all commands that are currently defined. It is useful in confirming the existence or exact name of a command that you want to modify, since eMatrix is case-sensitive.

```
list command [modified after DATE] NAME_PATTERN [select FIELD_NAME {FIELD_NAME}] [DUMP [RECORDSEP]] [tcl] [output FILENAME];
```

For details on the List statement, see [List Admintype Statement](#) in Chapter 1.

Use the list of all the existing commands along with the Print statement to determine the search criteria you want to change.

Use the Modify Command statement to add or remove defining clauses and change the value of clause arguments:

```
modify command NAME [MOD_ITEM {MOD_ITEM}];
```

NAME is the name of the command you want to modify.

MOD_ITEM is the type of modification you want to make. Each is specified in a Modify Command clause, as listed in the following command. Note that you need specify only the fields to be modified.

Modify Command Clause	Specifies that...
name NEW_NAME	The current command name is changed to the new name entered.
description VALUE	The current description value, if any, is set to the value entered.
icon FILENAME	The image is changed to the new image in the file specified.
label VALUE	The label is changed to the new value specified.
href VALUE	The link data is changed to the new value specified.
alt VALUE	The alternate text is changed to the new value specified.
code VALUE	The code associated with the command is replaced by the new code specified.
file FILENAME	The file that contains the command code is changed to the file specified.
add user NAME	The named user is granted access to the command.
add user all	All users are granted access to the command.
add setting NAME [STRING]	The named setting and STRING are added to the command.
remove user NAME	The named user access is revoked.
remove user all	Access to the command is revoked for all users.
remove setting NAME [STRING]	The named setting and STRING are removed from the command.
property NAME [to ADMINTYPE NAME] [value STRING]	The named property is modified.
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.

Each modification clause is related to the arguments that define the command. To change the value of one of the defining clauses or add a new one, use the Modify clause that corresponds to the desired change.

When modifying a command, you can make the changes from a script or while working interactively with MQL.

- If you are working interactively, perform one or two changes at a time to avoid the possibility of one invalid clause invalidating the entire statement.
- If you are working from a script, group the changes together in a single Modify Command statement.

Deleting a Command

If a command is no longer required, you can delete it using the Delete Command statements

```
delete command NAME:
```

NAME is the name of the command to be deleted.

When this statement is processed, eMatrix searches the list of defined commands. If the name is found, that command is deleted. If the name is not found, an error message is displayed. For example, to delete the command named "Convert," enter the following:

```
delete command "Convert";
```

After this statement is processed, the command is deleted and you receive an MQL prompt for another statement.

Using Macros and Expressions in Configurable Components

Many strings used in the definition of configurable Components (such as label values, hrefs, and settings) can contain embedded macros and select clauses. The \${ } delimiters identify macro names. Macros are evaluated at run-time. Macros for configurable components are available for directory specification. Some existing macros are also supported (refer to [Supported Macros and Selects](#) for more information).

Some strings can also include select clauses which are evaluated against the appropriate business object at run-time. The \$<> delimiters identify select clauses. Because the select clauses will generally use symbolic names, the clauses will be preprocessed to perform any substitutions before submitting for evaluation. The following example shows a macro being used in the href definition and another macro being used in the Image setting, as well as a select clause being used in the label definition of a tree menu (associated with a LineItem object):

```
MQL<2>print menu type_LineItem;
menu type_LineItem
description
label '$<attribute[attribute_EnteredName].value>'
href '${SUITE_DIR}/emxQuoteLineItemDetailsFS.jsp'
setting Image value ${COMMON_DIR}/iconSmallRFQLineItem.gif
setting Registered Suite value SupplierCentralSourcing
children
command SCSAttachment
command SCSCAttributeGroup
command SCSCHistory
command SCSSupplierExclusion
command SCSDA
nothidden
property original name value type_LineItem
property installed date value 02-28-2002
property installer value MatrixOneEngineering
property version value Verdi-0-0-0
property application value Sourcing
created Thu Feb 28, 2002 11:12:34 AM EST
modified Thu Feb 28, 2002 11:12:34 AM EST
```

The following example shows a typical business object macro being used in the label definition of a tree menu (associated with a Company object):

```
MQL<3>print menu type_Company;
menu type_Company
description
label '${NAME}'
href '${SUITE_DIR}/emxTeamCompanyDetailsFS.jsp'
setting Image value ${COMMON_DIR}/iconSmallOrganization.gif
setting Registered Suite value TeamCentral
children
command TMCBusinessUnit
command TMCLocation
command TMCPeople
nothidden
property original name value type_Company
property installed date value 02-28-2002
property installer value MatrixOneEngineering
```

```
property version value Verdi-0-0-0
property application value TeamCentral
created Thu Feb 28, 2002 11:31:57 AM EST
modified Thu Feb 28, 2002 11:31:57 AM EST
```

When using a macro, surround it with quotes to ensure proper substitution if a value contains spaces.

Supported Macros and Selects

The following sections provide lists of macros used in the configuration parameters of the administrative menu and command objects found in the AEF. These menu and command objects are used for configuring the Menus/Trees/Actionbars in the Value Chain Portfolio applications that use them. These are the only macros currently supported for use in any dynamic UI component.

Directory Macros

The following table provides the list of directory specific macros used in the configuration setting.

Directory Macros	
Macro Name	Description
<code> \${COMMON_DIR}</code>	To substitute the "common" directory below "ematrix" directory. The substitution is done with reference to any application specific directory and it is relative to the current directory.
<code> \${ROOT_DIR}</code>	To substitute the "ematrix" directory. The substitution is done with reference to any application specific directory below "ematrix" and it is relative to the current directory.
<code> \${SUITE_DIR}</code>	The macro to substitute the application specific directory below "ematrix" directory. The substitution is done based on the "Suite" to which the command belongs. and it is relative to the current directory.

Select Expression Macros

Select expression macros are defined as `$<SELECT EXPRESSION>`, where the select expression can be any valid MQL select statement. Select expression macros can be used in labels for configurable components and in expression parameters. These expressions are evaluated at runtime against the current business object ID and relationship ID that is passed in. Some examples include:

- `$<TYPE>`
- `$<NAME>`
- `$<REVISION>`
- `$<attribute[attribute_Originator].value>`

- \$<attribute[FindNumber].value>
- \$<from[relationship_EBOM].to.name>

29

Working With Menus

Menus Defined	646
Creating a Menu.....	647
Description Clause.....	647
Icon Clause.....	647
Label Clause.....	648
Href Clause.....	648
Alt Clause	648
Menu Clause.....	648
Command Clause.....	649
Setting Clause	649
Property Clause.....	649
Copying and/or Modifying a Menu	650
Copying (Cloning) a Menu Definition	650
Modifying a Menu	650
Deleting a Menu	653

Menus Defined

eMatrix Business Administrators can create new menu objects if they have the Menu administrative access. Menus can be used in custom Java applications. Before creating a menu, you must define the commands that it will contain, since commands are child objects of menus — commands are created first and then added to menu definitions, similar to the association between eMatrix types and attributes. Changes made in any definition are instantly available to the applications that use it. Menus can be designed to be toolbars, action bars, or drop-down lists of commands.

Creating a Menu

To define a menu from within MQL use the Add Menu statement:

```
add menu NAME [ADD_ITEM {ADD_ITEM}];
```

NAME is the name of the menu you are defining. Menu names cannot include asterisks.

You cannot have both a command and a menu with the same name.

ADD_ITEM is an Add Menu clause that provides additional information about the menu. The Add Menu clauses are:

description STRING_VALUE
icon IMAGE_PATH
label VALUE
href VALUE
alt VALUE
menu NAME { ,NAME }
setting NAME [STRING]
property NAME on ADMIN [to ADMIN] [value STRING]

Each clause and the arguments they use are discussed in the sections that follow.

Description Clause

The Description clause of the Add Menu statement provides general information about the function of the menu. Since there may be subtle differences between menus, you can use the description clause to point out the differences.

The description can consist of a prompt, comment, or qualifying phrase. There is no limit to the number of characters you can include in the description.

For example, if you were defining a menu named “Tools” you might write a Description clause similar to one of the following. Each menu might differ considerably.

```
description "Drawing Tools"  
description "Tools to figure cost"  
description "Shortcut Tools"
```

Icon Clause

Icons help users locate and recognize items by associating a special image with a menu. You can assign a special icon to the new menu or use the default icon. The default icon is used when in view-by-icon mode. Any special icon you assign is used when in view-by-image mode. When assigning a unique icon, you must use a .gif image file. Refer to [Icon Clause](#) in Chapter 1 for a complete description of the Icon clause.

.gif filenames should not include the @ sign, as that is used internally by eMatrix.

Label Clause

The Label clause of the Add Menu statement specifies the label to appear in the application in which the menu is assigned. For example, many desktop applications have a File menu.

Href Clause

The Href clause of the Add Menu statement is used to provide link data to the JSP. The Href link is evaluated to bring up another page. Many menus will not have an Href value at all. However, menus designed for the “tree” menus require an Href because the root node of the tree causes a new page to be displayed when clicked. The Href string generally includes a fully-qualified JSP filename and parameters, which can contain embedded macros and expressions for mapping to database schema. Refer to [Using Macros and Expressions in Configurable Components](#) for more details.

The syntax is:

```
href VALUE;
```

VALUE is the link data.

Alt Clause

The Alt clause of the Add Menu statement is used to define text that is displayed until any image associated with the menu is displayed and also as “mouse over text.”

The syntax is:

```
alt VALUE;
```

VALUE is the text that is displayed.

For example, you could use the following for a Tools menu:

```
alt "Tools";
```

Menu Clause

The Menu clause of the Add Menu statement is used to specify existing menus to be added to the menu you are creating. The menus will be displayed in the order in which they are added. Separate items with a comma.

The syntax is:

```
menu NAME { ,NAME };
```

NAME is the name of the menu you are adding.

Command Clause

The Command clause of the Add Menu statement is used to specify existing commands to be added to the menu you are creating. The commands will be displayed in the order in which they are added. Separate items with a comma.

The syntax is:

```
command NAME { ,NAME} ;
```

NAME is the name of the command you are adding.

For details on creating commands, see [Creating a Command](#) in Chapter 28.

Setting Clause

The Setting clause of the Add Menu statement is used to provide any name/value pairs that the menu may need. They can be used by JSP code, but not by hrefs on the Link tab. Refer to *Parameters and Settings for Commands* in the *Application Exchange Framework Guide* for appropriate settings for the type of menu you are creating. Also refer to [Using Macros and Expressions in Configurable Components](#) for more details.

The syntax is:

```
setting NAME [STRING] ;
```

For example, an image setting with the image name can be specified to display when the menu is used in a toolbar:

```
setting Image iconSmallMechanicalPart.gif ;
```

Property Clause

Integrators can assign ad hoc attributes, called Properties, to the menu. Properties allow associations to exist between administrative definitions that aren't already associated. The property information can include a name, an arbitrary string value, and a reference to another administration object. The property name is always required. The value string and object reference are both optional. The property name can be reused for different object references, that is, the name joined with the object reference must be unique for any object that has properties.

```
add menu NAME property NAME [to ADMINTYPE NAME] [value STRING] ;
```

For additional information on properties, see *Overview of Administration Properties* in Chapter 24.

Copying and/or Modifying a Menu

Copying (Cloning) a Menu Definition

After a menu is defined, you can clone the definition with the Copy Menu statement. Cloning a menu definition requires Business Administrator privileges, except that you can copy a menu definition to your own context from a group, role or association in which you are defined.

This statement lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy menu SRC_NAME DST_NAME [MOD_ITEM {MOD_ITEM}];
```

SRC_NAME is the name of the menu definition (source) to be copied.

DST_NAME is the name of the new definition (destination).

MOD_ITEMS are modifications that you can make to the new definition. Refer to the menu below for a complete list of possible modifications.

Modifying a Menu

The List Menu statement is used to display a list of all menus that are currently defined. It is useful in confirming the existence or exact name of a menu that you want to modify, since eMatrix is case-sensitive.

```
list menu [modified after DATE] NAME_PATTERN [select FIELD_NAME {FIELD_NAME}] [DUMP  
[RECORDSEP]] [tcl] [output FILENAME];
```

For details on the List statement, see [List Adminintype Statement](#) in Chapter 1.

Use the list of all the existing menus along with the Print statement to determine the search criteria you want to change.

Use the Modify Menu statement to add or remove defining clauses and change the value of clause arguments:

```
modify menu NAME [MOD_ITEM {MOD_ITEM}];
```

NAME is the name of the menu you want to modify.

MOD_ITEM is the type of modification you want to make. Each is specified in a Modify Menu clause, as listed in the following menu. Note that you need specify only the fields to be modified.

Modify Menu Clause	Specifies that...
name NEW_NAME	The current menu name is changed to the new name entered.
description VALUE	The current description value, if any, is set to the value entered.
icon FILENAME	The image is changed to the new image in the file specified.
label VALUE	The label is changed to the new value specified.
href VALUE	The link data information is changed to the new value specified.
alt VALUE	The alternate text is changed to the new value specified.
order menu NAME NUMBER	The order of the named menu item included in the menu is changed to the NUMBER specified. See Modifying the order of items in a Menu .
order command NAME NUMBER	The order of the named command item included in the menu is changed to the NUMBER specified. See Modifying the order of items in a Menu .
add menu NAME	The named menu is added to the menu.
add command NAME	The named command is added to the menu.
add setting NAME [STRING]	The named setting and STRING are added to the menu.
remove menu NAME	The named menu is removed from the menu.
remove command NAME	The named command is removed from the menu.
remove setting NAME [STRING]	The named setting and STRING are removed from the menu.
property NAME [to ADMINTYPE NAME] [value STRING]	The named property is modified.
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.

Each modification clause is related to the arguments that define the menu. To change the value of one of the defining clauses or add a new one, use the Modify clause that corresponds to the desired change.

When modifying a menu, you can make the changes from a script or while working interactively with MQL.

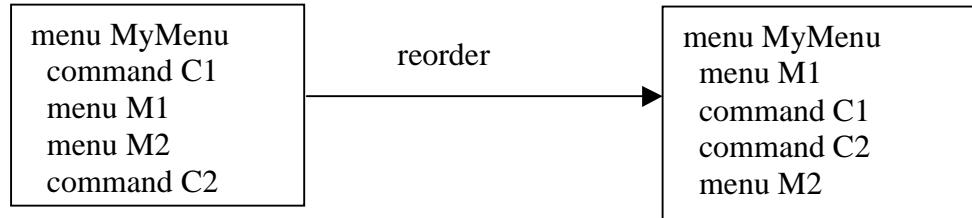
- If you are working interactively, perform one or two changes at a time to avoid the possibility of one invalid clause invalidating the entire statement.
- If you are working from a script, group the changes together in a single Modify Menu statement.

Modifying the order of items in a Menu

The `order` clause on the `modify menu` command is used to re-order menu items. For example, to place a new command (C3) in the third spot of an existing menu (MyMenu) that already has five items, you would issue the command:

```
modify menu MyMenu add command C3 order command C3 3;
```

Another example is given below showing the before and after effect of ordering.



In order to make this change you would have to issue the following command in MQL:

```
modify menu MyMenu order command C1 2 order command C2 3;
```

Deleting a Menu

If a menu is no longer required, you can delete it using the Delete Menu statement

```
delete menu NAME;
```

NAME is the name of the menu to be deleted.

When this statement is processed, eMatrix searches the list of defined menus. If the name is found, that menu is deleted. If the name is not found, an error message is displayed. For example, to delete the menu named “Toolbar” enter the following:

```
delete menu "Toolbar";
```

After this statement is processed, the menu is deleted and you receive an MQL prompt for another statement.

Working With Inquiries

Inquiries Defined.....	656
Creating an Inquiry	657
Description Clause.....	657
Icon Clause.....	657
Pattern Clause	657
Format Clause	658
Code Clause	658
File Clause.....	659
Argument Clause	659
Property Clause.....	659
Copying and/or Modifying an Inquiry	660
Copying (Cloning) an Inquiry Definition	660
Modifying an Inquiry.....	660
Evaluating an Inquiry	662
Deleting a Inquiry.....	663

Inquiries Defined

eMatrix Business Administrators can create new inquiry objects if they have the Inquiry administrative access. Inquiries can be evaluated to produce a list of objects to be loaded into a table in a JSP application. In general, the idea is to produce a list of business object ids, since they are the fastest way of identifying objects for loading into browsers. Inquiries include code, which is generally defined as an MQL `temp query` or `expand bus` command, as well as information on how to parse the returned results into a list of OIDs.

Creating an Inquiry

To define an inquiry from within MQL use the Add Inquiry statement:

```
add inquiry NAME [ ADD_ITEM {ADD_ITEM} ];
```

NAME is the name of the inquiry you are defining. Inquiry names cannot include asterisks. You must specify a unique name for each inquiry that you create. The name you choose is the name that will be referenced to evaluate this inquiry within a JSP.

ADD_ITEM is an Add Inquiry clause that provides additional information about the inquiry. The Add Inquiry clauses are:

description STRING_VALUE
icon IMAGE_PATH
pattern VALUE
format VALUE
code VALUE
file FILENAME
augument NAME [STRING]
property NAME on ADMIN [to ADMIN] [value STRING]

Each clause and the arguments they use are discussed in the sections that follow.

Description Clause

The Description clause of the Add Inquiry statement provides general information about the function of the inquiry. Since there may be subtle differences between inquiries, you can use the description clause to point out the differences.

The description can consist of a prompt, comment, or qualifying phrase. There is no limit to the number of characters you can include in the description.

Icon Clause

Icons help users locate and recognize items by associating a special image with an inquiry. You can assign a special icon to the new inquiry or use the default icon. The default icon is used when in view-by-icon mode. Any special icon you assign is used when in view-by-image mode. When assigning a unique icon, you must use a .gif image file. Refer to [Icon Clause](#) in Chapter 1 for a complete description of the Icon clause.

.gif filenames should not include the @ sign, as that is used internally by eMatrix.

Pattern Clause

The Pattern clause of the Add Inquiry statement indicates the expected pattern of the results of the evaluated code, and shows how the output should be parsed. It sets the desired field to an RPE variable or macro. Since inquiries are designed to produce a list of business objects, generally the macro that is set is OID.

When you execute a temp query in MQL, the business objects found are returned in a list that includes the type, name and revision, as well as any selectable information specified. For example, the following code:

```
MQL< >temp query bus Part * * select id dump;
```

would return a list like:

```
Part,PT-6170-01,1,21762.30027.65182.63525  
Part,PT-6180-01,1,21762.30027.50161.30295  
Part,PT-6190-01,1,21762.30027.56625.19298  
Part,PT-6200-01,1,21762.30027.37094.65388
```

To indicate that there are four fields that will be returned, delimited with a comma, and the last field is the OID, you would use the following pattern:

```
*,*,*,${OID}
```

For an expand bus command, even more information is output before the select fields:

```
MQL< >expand bus Person "Test Buyer" - from relationship "Assigned Buyer" select  
businessobject id dump |;  
1|Assigned Buyer|to|Buyer Desk|Buy 001|-|37819.19807.45300.63521
```

To parse this output, you need to indicate that the first six fields, delimited by "|", should be ignored, and the seventh field is the OID. You would use:

```
*|*|*|*|*|*|$ {OID}
```

Format Clause

The Format clause of the Add Inquiry statement defines what part of the output results should be saved in the inquiry's list. It references variables or macros specified in the Pattern, and can include delimiters.

The syntax is:

```
format VALUE;
```

VALUE is the part of the output results that should be saved in the inquiry's list.

For example:

```
format ${OID};
```

Code Clause

The Code clause of the Add Inquiry statement is used to provide the code to be evaluated to produce a list of one or more business objects.

The syntax is:

```
code VALUE;
```

VALUE is the code commands and statements.

The code provided is generally an MQL temp query or expand bus command that selects the found objects' ids. It can contain complicated where clauses as needed. For example:

```
temp query bus "Package" * *  
where  
  "('Project'==to[Vaulted Documents Rev2].businessobject.to[Workspace  
Vaults].businessobject.type)
```

```
&& ('${USER}'==to[Vaulted Documents Rev2].businessobject.to[Workspace  
Vaults].businessobject.from[Project Members].businessobject.to[Project  
Membership].businessobject.name)"  
select id dump |;
```

When macros are included in the code (\${USER} in example above), they should be surrounded by single or double quotes, in case the substitution contains a space. Quotes around both the macro in the code and the Argument when it contains a space ensures that the macro substitution is handled correctly.

File Clause

The Code for the inquiry does not need to be included in the Add Inquiry command itself. It can be written in an external editor.

The syntax is:

```
file FILENAME;
```

FILENAME is the name of the file that contains the code for the inquiry.

Argument Clause

The Argument clause of the Add Inquiry statement is used to provide any input arguments that the inquiry may need. Arguments are name/value pairs that can be added to an Inquiry as necessary to be used by the inquiry code. Depending upon how you write the code in both the Inquiry and the JSP, you may or may not use arguments.

The syntax is:

```
argument NAME [ STRING];
```

NAME is the name of the argument.

STRING is the input argument to be added. Include quotes if the value contains a space.

Quotes around both the macro in the code and the Argument when it contains a space ensures that the macro substitution is handled correctly.

Property Clause

Integrators can assign ad hoc attributes, called Properties, to the inquiry. Properties allow associations to exist between administrative definitions that aren't already associated. The property information can include a name, an arbitrary string value, and a reference to another administration object. The property name is always required. The value string and object reference are both optional. The property name can be reused for different object references, that is, the name joined with the object reference must be unique for any object that has properties.

```
add inquiry NAME property NAME [to ADMINTYPE NAME] [value STRING];
```

For additional information on properties, see *Overview of Administration Properties* in Chapter 24.

Copying and/or Modifying an Inquiry

Copying (Cloning) an Inquiry Definition

After an inquiry is defined, you can clone the definition with the Copy Inquiry statement. Cloning a inquiry definition requires Business Administrator privileges, except that you can copy a inquiry definition to your own context from a group, role or association in which you are defined.

This statement lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy inquiry SRC_NAME DST_NAME [MOD_ITEM {MOD_ITEM}];
```

SRC_NAME is the name of the inquiry definition (source) to be copied.

DST_NAME is the name of the new definition (destination).

MOD_ITEMS are modifications that you can make to the new definition. Refer to the inquiry below for a complete list of possible modifications.

Modifying an Inquiry

The List Inquiry statement is used to display a list of all inquiries that are currently defined. It is useful in confirming the existence or exact name of an inquiry that you want to modify, since eMatrix is case-sensitive.

```
list inquiry [modified after DATE] NAME_PATTERN [select FIELD_NAME {FIELD_NAME}] [DUMP  
[RECORDSEP]] [tcl] [output FILENAME];
```

For details on the List statement, see [List Adminintype Statement](#) in Chapter 1.

Use the list of all the existing inquiries along with the Print statement to determine the search criteria you want to change.

Use the Modify Inquiry statement to add or remove defining clauses and change the value of clause arguments:

```
modify inquiry NAME [MOD_ITEM {MOD_ITEM}];
```

NAME is the name of the inquiry you want to modify.

MOD_ITEM is the type of modification you want to make. Each is specified in a Modify Inquiry clause, as listed in the following inquiry. Note that you need specify only the fields to be modified.

Modify Inquiry Clause	Specifies that...
name NEW_NAME	The current inquiry name is changed to the new name entered.
description VALUE	The current description value, if any, is set to the value entered.
icon FILENAME	The image is changed to the new image in the file specified.
pattern VALUE	The pattern is changed to the new value specified.

Modify Inquiry Clause	Specifies that...
format VALUE	The format is changed to the new output results specified by value.
code VALUE	The code associated with the inquiry is replaced by the new code specified.
file FILENAME	The file that contains the inquiry code is changed to the file specified.
property NAME [to ADMINTYPE NAME] [value STRING]	The named property is modified.
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.

Each modification clause is related to the arguments that define the inquiry. To change the value of one of the defining clauses or add a new one, use the Modify clause that corresponds to the desired change.

When modifying an inquiry, you can make the changes from a script or while working interactively with MQL.

- If you are working interactively, perform one or two changes at a time to avoid the possibility of one invalid clause invalidating the entire statement.
- If you are working from a script, group the changes together in a single Modify Inquiry statement.

Evaluating an Inquiry

You can evaluate the Inquiry to determine if it will parse the output as you have designed the JSP to expect to receive it.

Use the `evaluate query` command to execute the inquiry's code, parse it, and display the generated list.

To override any specified Arguments, or include input that the inquiry may otherwise receive from the JSP, enter name/value pairs. Include only a space between multiple inputs, using quotes around values that contain spaces.

The syntax is:

```
evaluate inquiry NAME [NAME VALUE [NAME VALUE [...]]];
```

Deleting a Inquiry

If an inquiry is no longer required, you can delete it using the Delete Inquiry statements

```
delete inquiry NAME:
```

NAME is the name of the inquiry to be deleted.

When this statement is processed, eMatrix searches the list of defined inquiries. If the name is found, that inquiry is deleted. If the name is not found, an error message is displayed. For example, to delete the inquiry named “sub-assembly parts,” enter the following:

```
delete inquiry "sub-assembly parts";
```

After this statement is processed, the inquiry is deleted and you receive an MQL prompt for another statement.

Part IV:

eMatrix User Functions

31

Working With Context

Context Defined	668
Setting Context.....	669
Setting Context With Passwords	669
Setting Context Without Passwords	670
Setting Context With Disabled Passwords	671
Setting Context Temporarily	671

Context Defined

Setting *context* identifies the user to eMatrix, telling eMatrix the areas of access the current user maintains. For example, setting context to a person who is defined as a Business Administrator allows access to eMatrix Business Administrator functions such as adding a Type. In addition, a default vault is associated with context so that newly created objects are assigned to that user's typical vault (unless specified otherwise).

Personal settings that a user creates such as saved sets, queries, views and visuals (filters, cues, tips, toolsets and tables) are accessed only when context is set to that person.

Context is defined by persons' names and vault they are working on. As described in [Vault Defined](#) in Chapter 4, a vault defines a grouping of objects. For example, a vault may contain all the information and files associated with a particular project, product line, geographic area, or period of time.

MQL defaults the context as follows:

- For a newly installed (empty) eMatrix database, MQL assumes the context of "creator."
- If eMatrix has a person whose name matches the operating system user ID, the context is set to that person.
- If the operating system user ID does not match a person in eMatrix (and the user "guest" has not been deleted, made inactive, or assigned a password), the context is set to "guest."

If eMatrix does not (cannot) set a context as listed above, no default context is set.

Setting Context

Context identifies a person to eMatrix, indicates the type of person (such as a System Administrator), and optionally, provides security with a password. Any user can set context, but restrictions apply based on the type of person and password. For example, only a System Administrator can perform System Administrator functions.

Setting the context of a user implies that you are the user. Once the context is set, any statements you enter are subject to the same policies that govern the defined person. This is useful when you need to perform a large number of actions for a defined user.

For example, assume you want to include a person's files in the eMatrix database. When you include them, you want the person to maintain ownership. Also, you do not want to create objects the person cannot access or perform actions prohibited to the person. You need to *act as* the person when those files are processed. In other words, you want to identify yourself to eMatrix as the person in question so that the actions you take appear to have been done by the actual owner of the files.

Context is controlled with the Set Context statement which identifies a user to eMatrix by specifying the person name and vault:

```
set context [ITEM {ITEM}];
```

ITEM is a Set Context clause.

The Set Context clauses provide more information about the context you are setting. They are:

person PERSON_NAME
password VALUE
newpassword VALUE
vault VAULT_NAME

The Person clause can be replaced with the User clause. In both cases a defined person must be entered. A group or role is not allowed.

How you set the context varies based on whether the person definition includes a Password, No Password, or Disable Password clause. Each situation is described in the sections that follow.

Setting Context With Passwords

When a person is added to the eMatrix database, the Business Administrator may include a Password clause as part of the person's definition. This clause assigns an eMatrix password to the person. Once assigned, the password is required to access this person's context (unless the password is removed).

The password should be kept secret from all unauthorized users. If the defined person never shares its password with any other user, the effect is the same as using the Disable Password clause in the person's definition (refer to *Disable Password Clause* in Chapter 11). Use the following Set Context statement if a person is defined as having a password:

```
set context person PERSON_NAME password VALUE [vault VAULT_NAME];
```

PERSON_NAME is the name of a user defined in the eMatrix database.

VALUE is the password value assigned to the named person in the person definition that was created by the eMatrix Business Administrator.

VAULT_NAME is a valid vault defined in the eMatrix database.

In this statement, you must enter both a person name and the password associated with the person. If either value is incorrect, an error message is displayed. However, if you are the Business Administrator, you can bypass a defined password. If you are assigned a user type of Business Administrator, you can change your context to that of another person by entering the following statement:

```
set context person PERSON_NAME [vault VAULT_NAME];
```

For example, assume a person is defined as follows:

```
add person mcgovern
    fullname "Jenna C. McGovern"
    password PostModern
    assign role Engineer
    assign group "Building Construction"
    vault "High Rise Apartments";
```

If you are defined as a Full User and want to set your context to McGovern, you would enter:

```
set context user mcgovern password PostModern;
```

In this case, the Password clause must be included in the Set Context statement. If you are defined as a Business Administrator, you can set your context to mcgovern by entering:

```
set context person mcgovern;
```

No password is required even though a password was assigned. For more information on the different user types, see [Type Clause](#) in Chapter 11.

Changing Your Password

You can change your password as you set context using the keyword newpassword within the set context clause. Use this keyword with the user or person keywords and the password keyword. Enter the new password after the keyword newpassword. If the change is successful, context will be set as well. For example, the following MQL command will set context to the user mcgovern and change the user's password from "Jurassic" to "PostModern".

```
set context user mcgovern password Jurassic newpassword
PostModern;
```

Setting Context Without Passwords

When a person is defined with a [No Password Clause](#), anyone can set context to that person name. Since no password is required, the Set Context statement is:

```
set context person PERSON_NAME [vault VAULT_NAME];
```

For example, assume you want to access the business objects created by a person named MacLeod. To do this, you enter:

```
set context person macleod;
```

After this statement is executed, you have the same privileges and business objects as MacLeod.

Setting Context With Disabled Passwords

When a person is defined with a *Disable Password Clause*, the security for logging into the operating system is used as the security for setting context in eMatrix. When a user whose password is disabled attempts to set context in eMatrix, the system compares the user name used to log into the operating system with the person's eMatrix user name. If they match, the system sets context and a password is not necessary. If they do not match, the system denies access.

When Disable Password is chosen for a person, eMatrix changes any existing password. To reassign a password to such a person, create a new password for the person as you normally would.

Setting Context Temporarily

The context settings provided by the user at login time define what types of accesses that user has to eMatrix. Programs, triggers and wizards may be available to users who do not have appropriate access privileges to run them, so context must be changed within the program and then changed back to the original user's context when the program completes.

For example, a trigger program may need to switch context to perform some action which is not allowed under the current user context. It must then return to the original context to prohibit invalid access or ownership for subsequent actions.

Two MQL commands are available that are useful in scripts that require a temporary change to the session context.

Push Context command

The `push context` command changes the context to the specified person and places the current context onto a stack so that it can be recovered by a `pop context` command. `Push context` can also be issued with no additional clauses, in which case the current context would be placed on the stack, but the current context would be unchanged. The command syntax is:

```
push context [person PERSON_NAME] [password VALUE] [vault VAULT_NAME];
```

Pop Context command

The `pop context` command takes the last context from the stack and makes it current. The command syntax is:

```
pop context;
```

For example, a user needs to run a wizard that requires checkin access for a business object. Since the creator of the wizard does not know if all users who run the wizard will have checkin access, the wizard first changes context to assure checkin access:

```
push context user Taylor pass runwizard vault Denver;
```

The user Taylor is a person who has checkin access. It might even be a person definition created specifically for the purpose of running wizards.

The last thing the wizard does before the program terminates is to use the `pop context` statement to set context back to whatever it was before the wizard was run. The wizard creator does not have to know who is running the wizard or what the original context was.

Initialization Context Variable

There is an initialization file variable that controls whether context is restored or not at the termination of a program. If the `MX_RESTORE_CONTEXT` variable is set to `true`, the original user's context is restored after the program/wizard/trigger terminates. If set to `false`, any context changes made by a program/wizard/trigger will remain changed after the program terminates. This ensures that the appropriate action is taken if there is a failure within the program before the `pop context` command executes.

32

Working With IconMail

Overview of IconMail	674
Sending IconMail	675
To Clause	675
CC Clause	676
Subject Clause.....	676
Text Clause.....	676
Reading IconMail	677
Deleting an IconMail Message.....	678
Deprecated Function	678

Overview of IconMail

eMatrix offers an internal mail system, called IconMail, to enable eMatrix users to easily exchange business objects and text messages. This mail utility is similar to other electronic mail systems. However, as its name suggests, the icon of the discussed object is sent with the message, allowing the recipient to view or edit the object from within the mailbox.

While the IconMail utility is most often used from within eMatrix itself, you can access it through MQL. This enables the Business or System Administrator to send messages while performing other MQL statements. For example, the Administrator may use MQL to load external files for a group's use or assign a person to a role. Once the action is completed, while still in MQL the Administrator could send a mail message to notify the appropriate persons that the job is done.

Every user has access to IconMail unless it has been disabled in the user's person definition by the Business Administrator. For example, a user may create business objects and could use IconMail to notify the appropriate people about the existence and states of the objects.

The sender of mail does not know how it is received. It could be received as IconMail, e-mail, or both. With e-mail, the type, name, and revision of sent objects are added; but the e-mail recipient has no direct access to objects from the mail message.

Sending IconMail

You use the Send Mail statement to send mail to another eMatrix user:

```
send mail businessobject TYPE NAME REVISION [in VAULT] {ITEM};
```

OR :

```
send mail businessobject ID {ITEM};
```

TYPE is the name of the object's type.

NAME is the name of the defined business object.

REVISION identifies the object's revision status.

VAULT is the vault where the business object is held.

ID is the internal identification string assigned to the business object when it is created.

ITEM is one or more of these Send Mail statement clauses:

```
to USER_NAME { ,USER_NAME }
```

```
cc USER_NAME { ,USER_NAME }
```

```
subject VALUE
```

```
text VALUE
```

Although the clauses are optional, at least a Subject or Text clause is recommended.

Each clause and the arguments they use are discussed in the sections that follow.

To Clause

The To clause of the Send Mail statement identifies who should receive the message you are sending. It can contain a list of users:

```
to USER_NAME { ,USER_NAME }
```

USER_NAME is the name of a person, group, role, or association defined within the eMatrix database. Remember that user's names are case sensitive and must match exactly with their name definitions. If the name is not found, an error message is displayed.

Using the To clause enables you to specify persons, groups, roles, and associations. The fact that you can insert a role name or group name means that you not only have existing mailing lists but also can send a message to a person whose name you do not know but whose function it might be to process the information you have.

For example, you may want to send an announcement to everyone working on your project to inform them that certain materials are available for use:

```
send mail to "Vehicle Manufacturing"  
subject "Materials Now Available"  
text "The materials for the V34 Solar Vehicle are now available.  
For more information contact Mike Zimmerman at ext 511.";
```

When this message is sent, it will go to every user defined in the Vehicle Manufacturing group. If additional groups are needed, you can list them also by separating the names with a comma.

CC Clause

The CC clause of the Send Mail Statement circulates the mail to additional people. While the message may be intended for a single individual, this clause lets you notify others that the correspondence has taken place. Use the following syntax:

```
cc USER_NAME { , USER_NAME }
```

USER_NAME is the name of a person, role or group within the eMatrix database. Remember that user's names are case sensitive and must match exactly with their name definitions. If the name is not found, an error message is displayed.

Subject Clause

The Subject clause places a header on the mail message. This header is usually a short synopsis of the message's content. Use this syntax:

```
subject VALUE
```

VALUE is any string of characters following the MQL syntax rules.

By examining the subject header, the message reader should be able to identify the content or purpose of the mail message. For example, the following Subject clauses clearly indicate the content of the mail message:

```
subject "Testing of Building Fire Alarms this Weekend"
```

```
subject "Company Christmas Party December 19"
```

```
subject "Safety while using the new test equipment"
```

Depending on the purpose of the message you are sending, you may not include any text at all. It is possible to send one or more business objects or messages within the Subject clause only.

Text Clause

The Text clause contains the bulk of your mail message:

```
text VALUE
```

VALUE is any length character string you want to enter. The message may be short or quite lengthy.

When writing the text of your mail message, remember that you must enclose the entire content within either single quotes (' ') or double quotes (""). If your message includes apostrophes, enclose the message in double quotes. If your message includes double quotes, enclose the message in single quotes.

Reading IconMail

eMatrix IconMail is always sent to the server to which you're connected. However, in a distributed environment, you must point to the server from which you want to read mail.

You can read mail from MQL using the Print Mail statement:

```
print mail [server SERVER_NAME] [# | all];
```

SERVER_NAME is the name of the server from which you want to read mail messages.

is the message number to print. The message number equals the mail message oid (object id).

Use all to print all messages from the specified server, or from the default server if no server is specified.

The print mail statement without arguments prints all messages in the current user's mailbox.

Deleting an IconMail Message

After you are through with a mail message, you can delete it with the Delete Mail statement:

```
delete mail [server SERVER_NAME] [# | all];
```

SERVER_NAME is the name of the server from which you want to delete mail messages.

is the number of the specific message to be deleted. The message number equals the mail message oid (object id).

Use all to print all messages from the specified server, or from the default server if no server is specified.

When this statement is processed, eMatrix searches the list of existing mail messages. If the number is found, the mail message associated with that number is deleted along with any mail references to any business objects. If the number is not found, an error message is displayed.

If you want to delete all your IconMail, use the Delete Mail All statement:

```
delete mail all;
```

When this statement is processed, all your IconMail messages and the references to any business objects associated with them are deleted.

Note that only the mail reference to a business object is deleted—the business object remains untouched. Do not worry about inadvertently deleting an object when deleting an IconMail message.

Deprecated Function

The renumber mail command is no longer supported. As of version 9.0.6.0 IconMail messages are no longer numbered sequentially, but rather equal the mail message oids (object id). This change resolved a database contention/locking issue.

33

Working With Workflows

Overview of Workflows	680
Defining a Workflow	681
Description Clause.....	681
Image Clause.....	681
Vault Clause	683
Owner Clause	683
Attribute Clause	683
Managing Workflows.....	685
Starting a Workflow.....	685
Stopping a Workflow	685
Suspending a Workflow	685
Resuming a Workflow.....	685
Reassigning a Workflow	686
Planning Workflow Execution	686
Modifying a Workflow Definition	688
Modifying a Workflow Definition.....	688
Assigning Ad hoc Workflow Activity to Multiple Users	689
Deleting a Workflow	691

Overview of Workflows

Workflow is concerned with the automation of procedures, where documents, information and tasks are passed among participants according to a defined set of rules to achieve or contribute to an overall business goal.

Workflow *Instances* are based on Processes. The process definition contains a set of activities (tasks) connected with intelligent links that allow branching within the process. (See [Overview of Workflow Processes](#) in Chapter 21.)

A workflow instance is the representation of a single enactment of a process; it consists of workflow activities, also known as tasks. A process must exist before a workflow instance can be created.

When workflow is launched, the workflow instance and all the constituent activity instances are automatically created. The task assignments are dropped off in each user's IconMail inbox. A user completes the task and communicates the status of that task assignment to the workflow system. Based on the rules defined in the process definition, the workflow system routes the task to the next task performer(s), until the workflow is completed. Workflow owners are allowed to abort or suspend a workflow instance.

Defining a Workflow

Before a workflow can be created, a process definition must exist, since workflows are based on processes. For information on creating a process definition, see [Defining a Process](#) in Chapter 21.

A workflow is created with the Add Workflow statement:

```
add workflow PROCESS NAME [ADD_ITEM {ADD_ITEM}] ;
```

`PROCESS` is the name of the process on which the workflow is based.

`NAME` is the name you assign to the workflow. The workflow name is limited to 127 characters. The naming convention for workflow objects is similar to conventions for business objects. For additional information, refer to [Business Object Name](#) in Chapter 34.

`ADD_ITEM` is an Add Workflow clause that provides additional information about the Workflow. The Add Workflow clauses are:

description VALUE
image FILENAME
vault VAULT_NAME
owner USER_NAME
ATTRIBUTE_NAME VALUE {ATTRIBUTE_NAME VALUE}

All these clauses are optional. You can define a workflow by simply assigning a name to it. Each Add Workflow clause is described in detail in the sections that follow.

A validation check is run during workflow instance creation. The validate code is called during an `add workflow` and gives appropriate errors if the process on which the workflow is based is not valid.

Description Clause

The Description clause of the Add Workflow statement provides general information for you and the user about the workflow and the overall function of the workflow. There may be subtle differences between workflows; the description can point out the differences.

There is no limit to the number of characters you can include in the description. You can enter a string of any length. However, keep in mind that the description is displayed when the mouse pointer stops over the workflow in the Workflow chooser. Although you are not required to provide a description, this information is helpful when a choice is requested.

Image Clause

The Image clause of the Add Workflow statement associates a special image, called an ImageIcon, with a workflow. While it is optional, it can assist a user in locating a desired workflow. For example, you may have several different workflows within a vault. By having an ImageIcon of each workflow, a user can easily locate the workflow s/he desires by using the associated ImageIcon as a guide.

For example, if you have workflows to create parts, you might specify a GIF file of a drawing of each part as their ImageIcon. Then, as you were scanning a set of workflows using the eMatrix Navigator (not MQL), you could easily identify a particular workflow.

This would be true even if the workflows had names such as “New Part 41053” or “New Part 91617.”

While ImageIcons are very similar to icons, they require more display time and probably should be displayed only at specific times. Also, while an icon is associated with items such as policies, types, and formats, ImageIcons are associated only with business objects, persons and workflows. A workflow may or may not have an ImageIcon associated with it.

The GIF file needs to be accessible only during definition using the Add or Modify Workflow clause. Once an accessible file in the correct image type is used in the Workflow clause, eMatrix will read the image and store it with the workflow definition in the database. Therefore, the physical icon files do not have to be accessible for every eMatrix session. (If the file is not accessible during definition, eMatrix will be unable to display the image.)

To write an Image clause, you need the full directory path to the ImageIcon you are assigning. For example, the following statement assigns an ImageIcon of the actual part to the workflow:

```
add workflow "Create Part" "New Part 41590"  
    image $MATRIXHOME/demo/part41590.gif;
```

You can view an image with the View menu options or by setting the Session Preferences options in eMatrix Navigator.

Specifying redundant icons adds redundant information in the database that requires more work to display. When the default icon is desired, do not specify it because it requires more work to display.

The Image clause of the Add Workflow statement is optional unless there is a need to distinguish between workflows of the same type. If you do not specify an image, the default icon of the process is used.

Retrieving the Image

If you associate an image with a workflow using the Image clause, you can retrieve the image as a GIF file using the Image statement. The GIF file is placed in the MATRIXHOME directory, unless overridden by the optional clause, directory.

```
image workflow PROCESS NAME [directory DIRNAME] [file FILENAME] [verbose];
```

PROCESS is the process definition on which the workflow instance is based.

NAME is the name of the workflow instance from which you want to get the image.

DIRNAME is the full pathname where you want to save the image file. If no pathname is specified, the file is saved in the \$MATRIXHOME directory.

FILENAME is the name to be given to the image .gif file. If the file name is not specified, the file is given the name of the workflow with a .gif extension.

Using verbose prints the file name on the screen.

For example:

```
image workflow "Create Part" "New Part 41093";
```

retrieves the image of workflow “Create Part” “New Part 41093” and saves it to the \$MATRIXHOME directory with the name “New Part 41093.gif”.

Vault Clause

The Vault clause of the Add Workflow statement specifies the name of the vault where the workflow will reside.

eMatrix must know which vault is associated with the workflow. This element is optional because eMatrix will use the current vault (as set in the current context) as a default value. Your current vault will be used to contain the workflow being created. If the workflow you are creating should not be in the current vault, you must list the vault's name in the workflow's definition.

For example, assume you are in a vault named "Corporate." You decide to create a workflow with the specification "Purchase Req" "Order Materials." You could do this by entering the following statement:

```
add workflow "Purchase Req" "Order Materials";
```

Since a vault is not specified, eMatrix will use the current vault, "Corporate," to store the workflow. But you might rather have all workflows related to purchasing in the Finance vault. You could include a Vault clause:

```
add workflow "Purchase Req" "Order Materials"  
    vault "Finance";
```

In many ways, vaults resemble and operate like directories on computers. Your context is similar to your default directory. If the workflow (like a file) is not in the default directory, you must include the directory as part of the workflow definition.

Owner Clause

The Owner clause of the Add Workflow statement defines who the owner of the workflow will be. The Owner clause is optional. If you do not include one, MQL will assume the owner is the current user. The current user is defined by the present context. This means that the System and Business Administrators can create workflows for other users by first setting the context to that of the desired user and then creating the workflows, or by using the Owner clause.

If the user name you give in the Owner clause is not found, an error message will result. If that occurs, use the List User statement to check for the presence and spelling of the user name. Names are case-sensitive and must be spelled using the same mixture of uppercase and lowercase letters.

For example, the following workflow definition assigns the role "Jan Engineer" to a workflow titled "Module Design."

```
add workflow "Software Development" "Module Design" owner "Jan Engineer";
```

Attribute Clause

The Attribute clause of the Add Workflow statement allows you to assign a specific value to one of the workflow's attributes. A workflow's process may or may not have been designed to include process level attributes. If it was, you can assign a specific value to the attribute using the Attribute clause.

If you are unsure of either of the ATTRIBUTE_NAME or the VALUE to be assigned, you can use the Print Process and Print Attribute statements, respectively.

For example, the following Add Workflow statement assigns values to two attributes of the Purchase Part workflow, based on the Ordering process.

```
add workflow Ordering "Purchase Part"
    description "to purchase parts for manufacturing process"
    "Projected Duration" 5
    Quantity 16
```

Only the attributes associated with the process on which the workflow is based can be assigned values for the instance.

Managing Workflows

After a workflow is defined, workflow owners can change the disposition (state) of the workflow, reassign it to a new owner, or check the workflow path or status.

Starting a Workflow

Workflows can be started only by the owner of the workflow instance. When the workflow is started, it immediately sends out task assignments for the first activity in the workflow.

The following statement lets you start a previously-defined workflow:

```
start workflow PROCESS NAME;
```

PROCESS is the name of the process on which the workflow is based.

NAME is the name of the workflow you want to start.

Stopping a Workflow

When the workflow is stopped, all tasks which have been sent to users' IconMail inboxes are rescinded. Any users who are already working on tasks within the workflow receive an IconMail message stating that tasks have been rescinded.

The following statement lets you stop a workflow that is in progress:

```
stop workflow PROCESS NAME;
```

PROCESS is the name of the process on which the workflow is based.

NAME is the name of the workflow you want to stop.

Suspending a Workflow

When a workflow is suspended, no new tasks are assigned until the workflow is resumed.

A workflow is automatically suspended if an OR-split does not have a valid path to follow (that is, none of the transition expressions are satisfied).

The following statement lets you suspend a workflow that is in progress:

```
suspend workflow PROCESS NAME;
```

PROCESS is the name of the process on which the workflow is based.

NAME is the name of the workflow you want to suspend.

Resuming a Workflow

When a workflow is resumed, the workflow engine checks which tasks have been completed since the workflow was suspended and if necessary, proceeds to the next activity and assigns new tasks.

The following statement lets you resume a workflow that has been suspended:

```
resume workflow PROCESS NAME;
```

PROCESS is the name of the process on which the workflow is based.

NAME is the name of the workflow you want to resume.

Reassigning a Workflow

Only the workflow owner can reassign the workflow to a different user. Initially, the owner of the workflow is the person who creates it. That person can then reassign the workflow to a new owner. For example, you may have one person in the group who creates workflow instances, and then reassigns them to other users to manage.

The following statement lets you reassign a workflow to a new owner:

```
reassign workflow PROCESS NAME owner PERSON_NAME;
```

PROCESS is the name of the process on which the workflow is based.

NAME is the name of the workflow you want to reassign.

PERSON_NAME is the name of the person to whom you want to reassign the workflow. Note that a workflow can be reassigned only to a person, not to a group, role or association.

Planning Workflow Execution

The MQL select node command is available to find the name of the next node or activity that follows a specified node. This can be used to determine what paths a workflow would take once execution commences or to determine the current status of a workflow. To use the next selectable, the workflow must first be created from a defined process, but it does not need to be started.

Since the path a workflow takes depends mostly on how attribute values of the workflow and its component activities are set, you could use the selectable to determine the path that would be taken and adjust attribute values as needed. This allows you to ensure that the workflow instance will take the desired path before actually starting the workflow.

The print workflow command includes the following selectable for this purpose:

```
print workflow PROCESS WORKFLOW select node[WORKFLOW NODE].next;
```

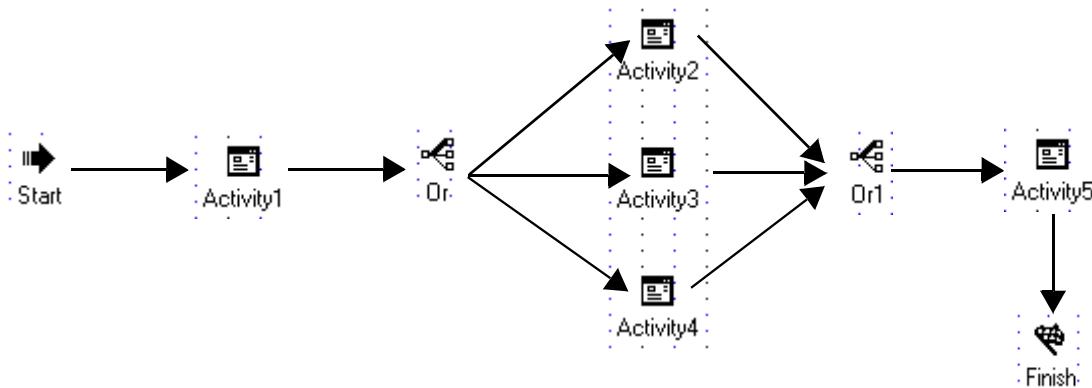
where:

PROCESS is the name of the process definition that is used for the workflow.

WORKFLOW is the name of the workflow instance.

NODE is the name of a node in the Process definition

As an example, suppose we have a process definition named simpleProcess whose flow graph is shown below.



First you could instantiate a workflow based on this process:

```
MQL<>add workflow simpleProcess simpleWorkflow vault KC;
```

To determine what is after the “Start” node you could use:

```
MQL<>print workflow 'simpleProcess' 'simpleWorkflow' select node[simpleWorkflow Start].next;
```

and MQL would return:

```
workflow simpleProcess simpleWorkflow
node[simpleWorkflow Start].next = simpleWorkflow Activity1
```

To determine what is after the Interactive Activity “Activity1” you could use:

```
MQL<>print workflow 'simpleProcess' 'simpleWorkflow' select interactive[simpleWorkflow Activity1].next;
```

and MQL would return:

```
workflow simpleProcess simpleWorkflow
interactive[simpleWorkflow Activity1].next = simpleWorkflow Or
```

To determine what is after the “Or” node you could use:

```
MQL<>print workflow 'simpleProcess' 'simpleWorkflow' select node[simpleWorkflow Or].next;
```

and MQL would return:

```
workflow simpleProcess simpleWorkflow
node[simpleWorkflow Or].next = simpleWorkflow Activity3
```

To determine the status of the next node you could use:

```
MQL<>print workflow 'simpleProcess' 'simpleWorkflow' select node[simpleWorkflow Start].next.status;
```

and MQL would return:

```
workflow simpleProcess simpleWorkflow
node[simpleWorkflow Start].next.status= completed
```

Modifying a Workflow Definition

Modifying a Workflow Definition

After a workflow is defined, you can change the definition with the Modify Workflow statement.

The following statement lets you add or remove defining clauses and change the value of clause arguments:

```
modify workflow PROCESS NAME [MOD_ITEM] {MOD_ITEM};
```

PROCESS is the name of the process on which the workflow is based.

NAME is the name of the workflow you want to modify.

MOD_ITEM is the type of modification you want to make.

You can make the following modifications. Each is specified in a Modify Workflow clause, as listed in the following table. Note that you only need to specify fields to be modified.

Modify Workflow Clause	Specifies that...
description VALUE	The current description, if any, is changed to the value entered.
image FILENAME	The current image is changed to use the new FILENAME entered.
owner USER_NAME	The current owner is changed to the new name entered.
attach businessobject OBJECTID	The named business object is attached to the workflow.
detach businessobject OBJECTID	The named business object is detached from the workflow.
ATTRIBUTE_NAME VALUE {,ATTRIBUTE_NAME VALUE}	The value of the named attribute is changed to the new value entered.
interactive TASK_NAME ATTRIBUTE_NAME VALUE {,ATTRIBUTE_NAME VALUE}	The attribute values belonging to the named interactive task are changed.
interactive TASK_NAME add user USER_NAME	The named task is assigned to the named user(s).
interactive TASK_NAME remove user USER_NAME {,USER_NAME}	The named task is removed from assignments for the named user(s).
interactive TASK_NAME reassignactivity user USER_NAME	The named task is reassigned to the named user.
interactive TASK_NAME completeactivity	The named interactive task is marked as completed.
interactive TASK_NAME suspendactivity	The named interactive task is suspended.
interactive TASK_NAME resumeactivity	The named interactive task is resumed.

Modify Workflow Clause	Specifies that...
interactive TASK_NAME overrideactivity	The named interactive task is overridden (skipped).
interactive TASK_NAME acceptactivity	The named interactive task is accepted by the owner of the current context.
automated TASK_NAME ATTRIBUTE_NAME VALUE { ,ATTRIBUTE_NAME VALUE }	The attribute values belonging to the named automated task are changed.
automated TASK_NAME completeautoactivity	The named automated task is marked as completed.
automated TASK_NAME suspendautoactivity	The named automated task is suspended.
automated TASK_NAME resumeautoactivity	The named automated task is resumed.
automated TASK_NAME reassignactivity user USER_NAME	The named task is reassigned to the named user.
automated TASK_NAME overrideautoactivity	The named automated task is overridden (skipped).

Assigning Ad hoc Workflow Activity to Multiple Users

Multiple users of any type can be added to the assignee list for an interactive activity, causing IconMail to be sent to all users specified. Ownership of the activity is not resolved until someone accepts the task. Using this assignee list allows the workflow owner to expand the number of IconMail recipients on an ad hoc basis, allowing multiple groups to be added during workflow execution. This means that two workflow instances that use the same process definition are able to distribute their IconMails to different sets of individuals. To summarize:

- An activity's assignees are those users who receive the IconMail for the activity. This includes the users assigned to the process definition, as well as users that are assigned to the workflow's activity.
- An activity owner is the person who accepts the task. The owner may not always be resolved, but once it is, it is always a single person user. Once a task is accepted, the IconMails to all others are rescinded.

You can add assignees to activities before the workflow has been started, while it is started, or by stopping (and restarting) it. (You cannot add assignees to activities when the workflow has been suspended or stopped.) When you restart a workflow, it returns to the beginning of the process.

To add and remove task assignees

The following command syntax is used to perform an ad hoc assignment:

```
modify workflow PROCNAME WORKFLOWNAME interactive ACTIVITYNAME add user USER { ,USER} ;
```

where USER is the name of a person, group, role, or association.

For example:

```
mod workflow "Simple Process" "Simple Workflow" interactive activity4 add user "Technical Writer", "Quality Engineer";
```

To remove a user from the list of assignees of a workflow activity, the syntax is:

```
modify workflow PROCNAME WORKFLOWNAME interactive ACTIVITYNAME remove user USER  
{ ,USER};
```

For example:

```
mod workflow "Simple Process" "Simple Workflow" interactive activity1 remove user  
"Technical Writer";
```

You can use the `print workflow` command to show all assignees, including both those defined in the process and those added by the workflow owner. For example:

```
print workflow "Simple Process" "Simple Workflow" select  
interactive[activity4].assignee;
```

Deleting a Workflow

If a workflow is no longer required, you can delete it with the Delete Workflow statement:

```
delete workflow PROCESS NAME;
```

PROCESS is the process on which the workflow is based.

NAME is the name of the workflow to be deleted. If the NAME contains embedded spaces, use quotation marks.

When this statement is executed, eMatrix searches the list of workflows. If the name is not found, an error message is displayed. If the name is found, the workflow is deleted.

34

Creating and Modifying Business Objects

Business Objects.....	694
Specifying a Business Object Name.....	695
Business Object Type	695
Business Object Name	695
Business Object Revision Designator.....	696
Object ID	697
Adding a Business Object	698
Description Clause.....	698
Image Clause.....	699
Vault Clause	700
Revision Clause	701
Owner Clause	701
Policy Clause	701
State Clause	702
Attribute Clause	703
Viewing Business Object Definitions	705
Print Businessobject Statement.....	705
Select Statements.....	706
Using select format.file.path	710
Copying and Modifying a Business Object.....	711
Copying/Cloning a Business Object	711
Modifying a Business Object	711
Creating a Revision of an Existing Business Object	718
Adding an Object to a Revision Sequence	719
Deleting a Business Object and Its Files.....	721

Business Objects

Business objects form the body of eMatrix. They contain much of the information an organization needs to control. Each object is derived from its previously-defined type and governed by its policy. Therefore, before users can create business objects, the Business Administrator must create definitions for the types (see [Working With Types](#) in Chapter 13) and policies (see [Working With Policies](#) in Chapter 17) that will be used. In addition, the users (persons, groups, and roles) must be defined in eMatrix before they can have access to the application (see [Working With Users](#) in Chapter 11).

When creating a business object, the first step for the user is to define (name) the object and assign an appropriate description and attribute values for it. File(s) can then be checked into the object and it can be manipulated by establishing relationships, moving it from state to state and perhaps deleting or archiving it when it is no longer needed. This chapter describes the basic definition of the object and its attributes. In the next chapter, relationships, connections, states, checking files in and out, and locking objects are described in more detail.

Specifying a Business Object Name

When you create or reference a business object, you must give its full business object name. The full business object name must contain three elements:

```
TYPE NAME REVISION
```

Each element must appear in the order shown. If any element is missing or if the values are given in the wrong order, the business object specification is invalid.

You can also optionally specify the vault in which the business object is held. When the vault is specified in this manner, only the named vault needs to be checked to locate the business object. This option can improve performance for very large databases.

```
TYPE NAME REVISION [in VAULT] | ID [in VAULT]
```

See [Adding a Business Object](#) later in this chapter for more information about additional elements.

The full business object specification includes TYPE NAME REV: the type from which the object was created, the object name—the user-supplied identifier that is associated with the definition and identifies the object to the end user(s)—and the revision.

In the sections that follow, each of the three required elements is discussed and sample values are given.

Business Object Type

The first element in a business object specification is the object's type. Every object must have a type associated with it. When you specify the object's type, remember that it must already be defined.

If the type name you give in the business object specification is not found, an error message will display. If this occurs, use the List Type statement (described in [List Admintype Statement](#) in Chapter 1) to check for the presence and spelling of the type name. Names are case-sensitive and must be spelled using the same mixture of uppercase and lowercase letters.

When you are assigning a type to a business object, note that all types have attributes associated with them. These attributes appear as fields when the object is accessed in eMatrix Navigator. These fields can then be filled in with specific values or viewed for important information.

For example, you might assign a type of “Metallic Component” to an object you are creating. With this type, you might have four attributes: type of metal, size, weight, and strength. If you use an Attribute clause in the add businessobject statement or modify the attributes, you can insert values that will appear whenever the object is accessed. If attributes are not specified when a business object is added or modified, the attribute defaults (if any) are used.

Business Object Name

The second element in a business object specification is the business object name. This consists of a character string that will be used to identify the business object and to reference it later. It should have meaning to you and guide you as to the purpose or contents of the business object. While the Description clause can remind you of an

object's function, it is time-consuming to have to examine each object to find the one you want. Therefore, you should assign a name that clearly identifies and distinguishes the object.

eMatrix is designed for you to use your exact business terminology rather than cryptic words that have been modified to conform to the computer system limitations. eMatrix has few restrictions on the characters used for naming business objects. Names are case-sensitive and spaces are allowed. You can use complete names rather than contractions, making the terminology in your system easier for people to understand. Generally, name lengths can be a maximum of 127 characters. Leading and trailing spaces are ignored.

You should avoid using characters that are programmatically significant to eMatrix, MQL, and associated applications. These characters include:

```
/ \ | * ^ ( ) [ ] { } = < > @ $ % & ! ? " ; : , §
```

In eMatrix, while commas are allowed in a business object's name, they are **not** recommended. In many statements a comma is the most common delimiter. Any of the characters listed above should be avoided when naming an object, a query or a set.

When specifying an existing business object, if the name you give is not found, an error message will result. Names are case-sensitive and must be spelled using the same mixture of uppercase and lowercase letters. If an error occurs, use the Temporary Query statement with wildcards to perform a quick search. For example, to find an object with a name beginning with the letters "HC" and unknown type and revision level, you could enter:

```
temporary query businessobject * HC* *
```

Use: the first * for the unknown type, the HC* for the name beginning with "HC", and the third * for the unknown revision level. The result would be all the objects beginning with "HC".

```
Product HC-430 A  
Product HC-470 B
```

Business Object Revision Designator

The third element in a business object specification is the revision label or designator. The revision must be specified if the object requires the revision label in order to distinguish it from other objects with the same name and type. Depending on the object's policy, revisions may or may not be allowed. If they are not allowed or a revision designator does not exist, you must specify "" (a set of double quotes) for MQL.

The ability (access privilege) to create revisions can be granted or denied depending on the object's state and the session context. When an object is revised, the revision label changes. This label is either manually assigned at the time the revision is created or automatically assigned if a revision sequence has been defined in the governing policy.

Revision sequences provide an easy and reliable way to keep track of object revisions. If the revision sequencing rules call for alphabetic labels, a revised object might have a label such as B or DD. If the Sequence clause in the policy definition specifies custom revision labels, you might see a label such as Unrevised, "1st Rev," "2nd Rev," and so on. In any case, the revision label you provide must agree with the revision sequencing rules. If it does not, an error message will result.

For example, the following are all valid business object specifications:

Component "NCR 1139" "
Drawing "Front Elevation" 2
Recipe "Spud's Fine Mashed Potatoes" IV
"Tax Record" "Sherry Isler" "second rev"

The first specification has no revision designator and must be specified as such. This might be because Component types cannot be revised under the governing policy. It might also be because this is the original object that uses a sequence where the first object has no designator.

For more information about revision sequences, see [Sequence Clause](#) in Chapter 17.

Object ID

When business objects are created, they are given an internal ID. As an alternative to TYPE NAME REV, you can use this ID when indicating the business object to be acted upon. The ID of an object can be obtained by using the print businessobject selectable “ID”. Refer to [Viewing Business Object Definitions](#) later in this chapter for more information on select statements.

When printing objects in a Loosely-Coupled Database (LCD) environment, it's a good idea to use the object ID instead of Type, Name, and Revision. Type, Name, and Revision may not be unique in this environment. Alternatively, you can use the in vault clause to unambiguously specify the object you want to use.

Adding a Business Object

Business objects are defined using the Add Businessobject statement:

```
add businessobject OBJECTID [ ITEM {ITEM} ];
```

OBJECTID is the OID or Type Name Revision of the business object.

ITEM is an Add Businessobject clause:

description VALUE
image FILENAME
vault VAULT_NAME
owner USER_NAME
policy POLICY_NAME
state STATE_NAME schedule DATE
ATTRIBUTE_NAME VALUE

The Policy clause is required when adding a new business object. It specifies the policy that will govern this object.

The Vault clause indicates the name of the vault where the object will reside. If the vault is not specified, the business object is created in the current context vault setting. The vault must be specified only if you want the business object to be created in another vault.

All Add Businessobject clauses provide information about the business object being defined. Only the Policy clause is required. You will learn more about each Add Businessobject clause in the sections that follow.

Description Clause

The Description clause of the Add Businessobject statement provides general information to both you and the user about the information associated with this business object. When a user creates a business object, this description will guide him/her to understand the information, function, or files associated with the object. There may be subtle differences between business objects—the Description clause enables you to point out these differences to the user.

For example, assume you have two versions of a project that you decide to develop concurrently. The resulting two business objects might have similar names such as “Solar Vehicle X29” and “Solar Vehicle X30.” Even if the reason for the different versions is contained elsewhere in business object’s contents, you can specify the reason in the Description clause. For example:

```
add businessobject "Energy Efficient Vehicle" "Solar Vehicle X29" A
  description "Uses traditional batteries";
  policy "Solar Vehicle"
add businessobject "Energy Efficient Vehicle" "Solar Vehicle X30" A
  description "Uses prototype batteries";
  policy "Solar Vehicle"
```

When you create the business object, you should assign a name that has meaning to both you and the user. In this example, the names of the two business objects are nearly the same. Without the Description clauses, you might inadvertently forget which project is which. Also, someone who is unfamiliar with the projects may not know the name assigned to the project they want. Therefore, while the Description clause is optional, it is recommended.

Image Clause

The Image clause of the Add Businessobject statement associates a special image, called an ImageIcon, with a business object. While it is optional, it can assist a user in locating a desired object. For example, you may have several different components, blueprints, or designs within a vault. By having an ImageIcon of each item associated with its object name, a user can easily locate the object s/he desires by using the associated ImageIcon as a guide.

For example, you might specify a GIF file of a drawing of each object as their ImageIcon. Then, as you were scanning a set of business objects using the eMatrix Navigator (not MQL), you could easily identify a particular object. This would be true even if the objects had names such as “J391” or “X19.”

While ImageIcones are very similar to icons, they require more display time and probably should be displayed only at specific times. Also, while an icon is associated with items such as policies, types, and formats, ImageIcones are associated only with objects and persons. An object may or may not have an ImageIcon associated with it. Excluding an ImageIcon from this object will not affect other objects that are created with the same type, policy, state, and owner. You can even have two revisions of an object with different ImageIcones since each revision represents a different object.

The GIF file needs to be accessible only during definition using the Add or Modify Businessobject clause. Once an accessible file in the correct image type is used in the Businessobject clause, eMatrix will read the image and store it with the object definition in the database. Therefore, the physical icon files do not have to be accessible for every eMatrix session. (If the file is not accessible during definition, eMatrix will be unable to display the image.)

To write an Image clause, you need the full directory path to the ImageIcon you are assigning. For example, the following statement assigns an ImageIcon of the actual vehicle to the business object:

```
add businessobject "Energy Efficient Vehicle" "Solar Vehicle X29" A
  policy "Solar Vehicle"
  description "Uses traditional batteries"
  image $MATRIXHOME/demo/vehicleX29.gif;
```

You can view an image with the View menu options or by setting the Session Preferences options in eMatrix Navigator.

Specifying redundant icons adds redundant information in the database that requires more work to display. When the default icon is desired, do not specify it.

The Image clause of the Add Businessobject statement is optional unless there is a need to distinguish between objects of the same type. If you do not specify an image, the default icon of the type is used.

Retrieving the Image

If you associate an image with a business object using the Image clause, you can retrieve the image as a GIF file using the Image statement. The GIF file is placed in the \$MATRIXHOME directory, unless overridden by the optional clause, directory.

```
image businessobject OBJECTID [directory DIRNAME] [file FILENAME] [verbose];
```

OBJECTID is the OID or Type Name Revision of the business object instance from which you want to get the image.

DIRNAME is the full pathname where you want to save the image file. If no pathname is specified, the file is saved in the \$MATRIXHOME directory.

FILENAME is the name to be given to the image .gif file. If the file name is not specified, the file is given the name of the object with a .gif extension.

Using verbose prints the file name on the screen.

For example:

```
image bus Assembly 12345 0;
```

retrieves the image of object Assembly 12345 0 and saves it to the \$MATRIXHOME directory with the name 12345.gif.

Vault Clause

The Vault clause of the Add Businessobject statement specifies the name of the vault where the object will reside. If you include a Vault clause in your definition, it must be placed after the Policy clause and before any other Add Businessobject clauses.

eMatrix must know which vault is associated with the business object. This element is optional because if not explicitly stated, eMatrix will use the current vault (as set in the current context) as a default value. If the object you are creating should not be in the current vault, you must include the vault's name in the object's definition.

For example, assume you are in a vault named "Car Loans." You decide to create an object with the specification "Car Loan" "Alan Broder." You could do this by entering the following statement:

```
add businessobject "Car Loan" "Alan Broder" A  
policy "Bank Loans";
```

Since a vault is not specified, eMatrix will use the current vault, "Car Loans," to store the business object named "Alan Broder." But what if Mr. Broder also had a mortgage with the same company and you were in the "Mortgages" vault instead of the "Car Loans" vault? You would have to include a Vault clause:

```
add businessobject "Car Loan" "Alan Broder" A  
policy "Bank Loans"  
vault "Car Loans";
```

In many ways, vaults resemble and operate like directories on computers. Your context is similar to your default directory. If the object (like a file) is not in the default directory, you must include the directory as part of the object name.

Revision Clause

The Revision clause of the Add Businessobject statement adds the revision label or designator for the business object. When a user creates a business object the revision level is required to distinguish two different versions of the same object. Depending on your context and the current state of the original object (if you are defining a revision for an existing object), you may not be able to assign a revision level when you define it. It may be automatically generated or not allowed. See [Business Object Revision Designator](#) earlier in this chapter for more information and examples of revision designators.

Owner Clause

The Owner clause of the Add Businessobject statement defines who the owner of the business object will be. The Owner clause is optional when defining a business object. If you do not include one, MQL will assume the owner is the current user, which is defined by the present context. This means that the System and Business Administrators can create objects for other users by first setting the context to that of the desired user and then creating the desired objects, or by using the Owner clause.

The owner of an object can be assigned special access in the policy. The user who is assigned ownership of an object has access privileges defined in the Owner subclause of the policy. That user can be an individual, a group, or a role.

Access privileges for the owner of an object are assigned in the Owner subclause of the policy. The owner can be an individual, a group, or a role.

If the user name you give in the Owner clause is not found, an error message will result. If that occurs, use the List User statement to check for the presence and spelling of the user name. Names are case-sensitive and must be spelled using the same mixture of uppercase and lowercase letters.

For example, the following object definition assigns the role “Software Developer” as the owner of a business object titled “Graphics Display Routine:”

```
add businessobject "Computer Program" "Graphics Display Routine" A
    policy "Software Development"
    description "Routine for displaying information on a color monitor"
    owner "Software Developer";
```

In this example, the administrator may not have a specific name of a user to assign to the business object. Therefore, the name of a role is used. All persons assigned the role of Software Developer can access the object as the owner. At a later time, a Person can be reassigned as the owner according to the reassign access rules specified in the governing policy.

Policy Clause

The Policy clause of the Add Businessobject statement assigns a policy to the business object. A policy controls a business object. It specifies the rules that govern access, approvals, lifecycle, versioning and revisioning capabilities, and more. If there is any question as to *what you can do* with a business object, it is most likely answered by looking at the object’s policy. Specifically, a policy defines the following information:

- The types of objects the policy will govern.
- The types of formats that are allowed for file checkin and the default format.
- Where and how checked-in files are managed.
- How revisions will be labeled.
- The number and order of each object state.

- The restrictions, if any, associated with each object state.

Since this information is required for a business object to be usable, a Policy clause must be included in the business object definition. If you are unsure about the policy, you should examine the policy definition with the Print Policy statement (see [Working With Policies](#) in Chapter 17).

The policy specified must be defined to govern the type of business object being created.

If the policy name you give in this clause is not found, an error message results. If that occurs, use the List Policy statement to check for the existence and spelling of the policy name.

For example, the following object definition assigns the “Software Development” policy to a business object titled “Graphics Display Routine:”

```
add businessobject Routine "Graphics Display Routine" 1
  policy "Software Development";
  description "Routine for displaying information on a color monitor"
```

After this statement is processed, the business object will be created and the “Software Development” policy will control who can access the object and the object’s lifecycle states.

State Clause

The lifecycle of a business object is assigned as part of the policy and consists of a series of object states. A state identifies a stage in the lifecycle of an object. Depending on the type of object involved, the lifecycle might contain only one state or many states. The states control the actual object instances and specify what can be done with an object after it is created. Each state defines who will have access to the object in that state, what type of access is allowed, whether or not the object can be revised, whether or not files within the object can be revised, and the conditions required for changing state.

The State clause of the Add Businessobject statement is used to specify the scheduled target date for this object in a particular state. The State clause is optional. It is not necessary to assign a date for any particular state. There may be situations where the change from one state to another may occur at any time. For example, if automobile insurance is contained within an object, the object might change when an accident occurs. Since accidents are not usually planned, no schedule date can be assigned. If, however, you have a state for policy renewal, the date can be scheduled since you know when the policy renewal is due.

Note that this is a target date only. The object will not automatically enter that state on the given date. The object can only be promoted after all required conditions are met—this date will not influence those conditions. The date is intended for informational purposes only.

When specifying the name of the state to which you want to assign a date, you must make sure it is a valid name. If the state name given in the State clause is not found in the policy you are assigning to the object, an error message will result. If that occurs, use the Print Policy statement to check for the existence and spelling of the state name.

When specifying the date within a State clause, you must use the date/time formats defined in your initialization file. If nothing is set there, the defaults are used. Consult the *eMatrix Installation Guide* for more information about defining date and time formats.

The default settings for date/time formats include the year, month, day, hour, minute, and/or second. Abbreviations or full-words are acceptable for the day of the week and/or month.

In the following example, the day of the week is optional:

```
Wed Feb 15, 1999
```

Another way to enter this date is:

```
2/15/99
```

The date also can include the time of day. In the following example, the meridian and timezone information are optional:

```
01:30:00 PM EST
```

When you enter both the date and time, the time should follow the date. For example:

```
February 1, 1999 12:52:30 GMT
```

The actual date/time is calculated based on the current time and date obtained from your system clock. The acceptable range is January 1, 1902 to December 31, 2037.

For example, the following object definition will schedule the object to be in the “Re-evaluation and Renewal” state on June 30, 2002.

```
add businessobject "Homeowner's Insurance" "Hebron Family" B
  policy "Primary Homeowners"
  description "Homeowner's Insurance for Hebron's Primary Residence"
  image $MATRIXHOME/demo/house.gif
  state "Re-evaluation and Renewal" schedule "June 30, 2002";
```

In this example, you want to remind the agent of when the homeowner’s insurance policy should be renewed. By inserting this date, an agent might promote the object to the “Re-evaluation and Renewal” state even if the date is June 26 (since it is close enough to the target date of June 30th.)

Attribute Clause

The Attribute clause of the Add Businessobject statement allows you to assign a specific value to one of the object’s attributes. As stated earlier, you must assign a type to any object being created. An object’s type may or may not have attributes associated with it. If it does, you can assign a specific value to the attribute using the Attribute clause.

If you are unsure of either of the ATTRIBUTE_NAME or the VALUE to be assigned, you can use the Print Type and Print Attribute statements, respectively.

For example, assume you are defining an object of type “Shipping Form” by using the following Add Businessobject statement:

```
add businessobject "Shipping Form" "Lyon's Order" A
  policy "Shipping";
  description "Shipping Form for the Lyon's Order"
  image $MATRIXHOME/demo/label.gif
```

Only the attributes associated with an object’s type can be assigned values for the instance.

If you were to examine the definition for the type “Shipping Form,” you might find it has three attributes associated with it called “Label Type,” “Date Shipped,” and “Destination

Type.” When this type is assigned to the object called “Lyon’s Order,” these attributes are automatically associated with the object. It is up to you as the user to assign values to the attributes.

If you do not assign values, they remain blank or the default is used if there is one. To assign values, you can insert an Attribute clause into the object definition.

When you are specifying an attribute value, be sure the value is in agreement with the attribute definition. In other words, only integer values are assigned to integer attributes, character string values are assigned to character string attributes, and so on. Also, if the attribute has a range of valid values, the value you give must be within that range.

You can use the Print Attribute statement to examine an attribute’s definition. For example, assume you are unsure of the definition of “Label Type.” If you examine the attribute definition, you might see that it is a character string value with no predefined range. With this knowledge, you can insert an Attribute clause to define a value similar to the following:

```
"Label Type" "Overnight Express"
```

String attributes (as well as description fields) have a limit of 2,048 KB. If you expect to enter more data, consider checking in a file instead.

If you want to assign values to each of the three attributes, you can write the object definition as:

```
add businessobject "Shipping Form" "Lyon's Order" A
    policy "Shipping"
    description "Shipping Form for the Lyon's Order"
    "Label Type" "Overnight Express"
    "Date Shipped" 12/22/1999
    "Destination Type" "Continental U.S.";
```

With this definition, each attribute associated with the type “Shipping Form” will have a specific value and those values can be viewed whenever the “Lyon’s Order” object is accessed.

Viewing Business Object Definitions

You can view the definition of a business object at any time by using the Print Businessobject statement and the Select Businessobject statement. These statements enable you to view all the files and information used to define the business object. The system attempts to produce output for each select clause input, even if the object does not have a value for it. If this is the case, an empty field is output.

There are four forms for the Print Businessobject statement and the Select Businessobject statement:

```
print businessobject OBJECTID [select SELECTABLE] [DUMP ["SEPARATOR_STR"]] [output  
FILENAME] [sortattributes];  
  
print businessobject selectable;
```

OBJECTID is the OID or Type Name Revision of the business object that you want to view.

SELECTABLE specifies a subset of the business object contents.

DUMP allows you to insert formatting data into the printed information; SEPARATOR_STR is the character used to separate business object details. A comma is often used as the separator string.

FILENAME identifies a file where the print output is to be stored.

Print Businessobject Statement

The Print Businessobject statement is the common form of the Print statement. Without any of the Print statement options, this statement appears as:

```
print businessobject OBJECTID;
```

OBJECTID is the OID or Type Name Revision of the object you want to print.

If the OBJECTID you give is not found, an error message results. If this occurs, use the List Businessobject statement to check the spelling of the business object name and to be sure it exists.

The Print Businessobject statement is similar to using the object inspector in eMatrix Navigator (desktop version only).

When this statement is processed, MQL displays all information that makes up the named business object's definition. This information appears in alphabetical order according to names of the object's fields. For example, you could obtain the definition for the "Shipping Form" "Lyon's Order" A business object by entering:

```
print businessobject "Shipping Form" "Lyon's Order" A;
```

Unlike the other Print statements, the Print Businessobject statement uses three optional Print statement clauses:

- Select clause
- Dump clause
- Tcl clause
- Output clause

The optional forms of the Print Businessobject statement enable you to specify the information you want to retrieve from a business object. This subset is created by identifying the desired fields of the business object. The field contents can be prepared for formatting and stored in an external file. But which fields can you print and how do you specify them? The Print Businessobject Selectable and statement addresses these questions. This statement enables you to view a listing of field choices and specify those choices, as described in the following sections.

When printing objects in a Loosely-Coupled Database (LCD) environment, it's often a good idea to use the object ID instead of Type, Name, and Revision. Type, Name, and Revision may not be unique in this environment. Alternatively, you can use the in vault clause to unambiguously specify the object you wish to use.

Select Statements

Select statements enable you to view a listing of the field names that make up the business object definition. Each field name is associated with a printable value. By selecting and listing the field names that you want, you can create a subset of the object's contents and print only that subset. The system attempts to produce output for each select clause input, even if the object does not have a value for it. If this is the case, an empty field is output.

Select can be used as a clause of the print businessobject and expand businessobject statements. (See *Displaying and Searching Business Object Connections* in Chapter 35 for more information).

Reviewing the List of Field Names

The first step is to examine the general list of field names. This is done with the statement:

```
print businessobject selectable;
```

The Selectable clause is similar to using the ellipsis button in the graphical applications—it provides a list from which to choose.

When the statement above is processed, MQL will list all selectable fields with their associated values. This statement might produce a listing such as:

```
business object selectable fields:  
  name  
  description  
  revision  
  originated  
  modified  
  lattice.*  
  owner.*  
  grantor.*  
  grantee.*  
  granteeaccess  
  granteesignature.*  
  grantkey  
  policy.*  
  type.*  
  attribute[].*  
  default.*  
  format[].*  
  current.*  
  state[].*  
  revisions[].*  
  previous.*  
  next.*  
  first.*  
  last.*  
  history[].*  
  relationship[].*  
  to[].*  
  from[].*  
  toset[].*  
  fromset[].*  
  exists  
  islockingenforced  
  vault.*  
  locked  
  locker.*  
  id  
  method.*  
  search.*  
  workflow[].*
```

Notice that some of the fields are followed by square brackets and/or a ". *". The asterisk denotes that the field can be further expanded, separating the subfields with a period. If only one value is associated, the name appears without an asterisk. For example, the Name and Description fields each have only a single value that can be printed. On the other hand, a field such as type has other items that can be selected. If you expand the Type field, you might find fields such as type.name, type.description, and type.policy. This means that from any business object, you could select a description of its type and find out other valid policies for it:

```
print businessobject Drawing 726590 B select type.description type.policy;
```

The above may output something like this:

```
business object Assembly 726590 B
    type.description = Assembly of parts
    type.policy = Production
    type.policy = Production Alternative
```

All items that can be further expanded have a default subfield that is used when that field is specified alone. For example, selecting type is the same as selecting type.name, because the default for types is the type name. Refer to *Appendix A, Select Expressions* in the *eMatrix Navigator Guide* for tables of the expandable items with default values and examples of Select Expression usage.

The use of square brackets in the above selectable list indicates that the assigned name must also be included as part of the selection. For example, an object may have several attributes. To specify the attribute you want, you must give the assigned name of the attribute as part of the field name.

```
print bus Component 45782 A select attribute[Color];
```

The print bus command can provide grantor, grantee, granteeaccess, and granteesignature information. If there is no grantor and grantee, <none> will be specified. These items are selectable for use with print bus, expand bus, print set, or expand set commands. For example:

```
print bus Assembly SA-4356 A select grantor grantee granteeaccess granteesignature;
```

This command will print the business object, showing the grantor, grantee, grantee accesses, and true or false for granteesignature.

Selecting Field Names for Printing

Once you have identified names of the fields that can be printed, you can select them.

```
print OBJECT_ID select FIELD_NAME {FIELD_NAME};
```

FIELD_NAME can be a string, list of strings, or wildcard character (*) that represents a printable field value associated with a business object. Each field name obeys the following syntax:

```
FIELD_NAME [ASSIGNED_NAME_PATTERN].SUBFIELD_PATTERN
```

ASSIGNED_NAME_PATTERN is the name assigned to the field when it is created. This name, when required, must be within square brackets.

SUBFIELD_PATTERN is the specification of another field. This field is a part of the larger field specification. All subfield names are preceded by a period and often correspond to the clauses that make up a field definition.

For example:

This select statement is added as a clause to the Print Businessobject statement:

```
print businessobject Drawing 726596 B select name description type.name;
```

This yields:

```
business object Drawing 726596 B
  name = 726596
  description = Piston Assembly
  type.name = Drawing
```

Notice that the fields appear in the order they were specified in the Select clause.

Dump Clause

When printing selected objects, notice that the returned data includes the field names. If you do not want to print the field names, include the Dump clause. For example:

```
print businessobject Drawing 726596 B select name description type.name dump;
```

This might yield:

```
726596,Piston Assembly,Drawing
```

Field values are separated by commas. If you want some other field separator, you can include a separator string in the Dump clause in the Print Businessobject statement. For example:

```
print businessobject Drawing 726596 B select name description type.name dump ":::";
```

This statement yields:

```
726596::Piston Assembly::Drawing
```

Using Select clauses with the Dump clause

Use of the Dump clause requires explicitly specified select clauses. MQL print commands using dump without explicitly specified select clauses are not supported.

To be specific, this refers to print commands of the following forms

```
print bus T N R dump;
```

```
print admintype dump; (where admintype is type, attribute, relationship,
person....)
```

These commands, which omit specific select clauses, will “remember” the select clauses that were used in the most recent print command for the given object type (bus, type, attribute, relationship, person,...) in the current eMatrix session, and print the data for those clauses only.

When used in creating a program, these commands may result in an implicit, and usually unintended, dependency on other program objects, as the results of these commands could be affected by print xxx select commands that were executed in completely unrelated programs.

Tcl Clause

Use the Tcl clause after the dump clause and before the output clause to return the results in Tcl list format. This facilitates the parsing of output from MQL select commands within Tcl code since the built-in list handling features of Tcl are used directly on the results. For more information, see [Tcl Format Select Output](#) in Chapter 1.

Output Clause

The results of a print select statement can be saved to a text file with the use of the output clause. For example:

```
print businessobject Component 12345 A select name description  
dump output c:\description.txt;
```

As shown in the above example the path can be expressed as part of the file name. Otherwise the file will be placed in the \$MATRIXHOME directory.

If the `output` keyword is used on a `print businessobject` statement without a `select` keyword included, the select clauses that were last used are remembered and applied to the statement. If you require different information than you previously selected, include the `select` keyword.

Sortattributes Clause

When printing business objects, the Sortattributes clause causes the command to list attributes in the same order as in eMatrix.

Using select format.file.path

Output from the business object selectable format.file.path changed between version 8 and 9. The difference in the output is:

V8 and prior: `c:\mydocs\testfile.txt`
V9: `/c/mydocs/testfile.txt`

If an integration requires the older format of the output, the following code can be used to reformat the new style data:

```
% set fpath [mql print bus TestType what 1 select format.file.path dump]  
/C/WINNT/Profiles/User/Personal/sqlnet.log  
% regsub -all /C/ $fpath C:/ fpath  
1  
% regsub -all / $fpath \ fpath  
5  
% puts $fpath  
C:WINNTProfilesUserPersonalsqlnet.log
```

Copying and Modifying a Business Object

Copying/Cloning a Business Object

After a business object is added and its values defined, you can copy or clone the object with the Copy Businessobject statement. This statement lets you duplicate a business object's defining clauses and change some of the values at the same time. You might first view the object's contents using the Print Businessobject statements just described in [Viewing Business Object Definitions](#). Then you can use the Copy Businessobject clauses listed below to modify the values you want to change. You should recognize these clauses since they are the same in the Add Businessobject statement.

```
copy businessobject OBJECTID to NAME REVISION [ ITEM {ITEM} ];
```

OBJECTID is the OID or Type Name Revision of the existing business object.

NAME is the name for the new business object. If you use the same type, name and revision as the original business object, an error will occur and the business object will not be copied.

REVISION is the revision label or designator. If a revision is desired, it should be specified. Otherwise, it is " " (a set of double quotes).

ITEM is a Copy Businessobject clause. For each value you want to change in the new business object, you specify the new value with the appropriate clause.

description VALUE
image FILENAME
vault VAULT_NAME
name NAME [revision REVISION]
owner USER_NAME
policy POLICY_NAME
state STATE_NAME schedule DATE
type TYPE_NAME
ATTRIBUTE_NAME VALUE

Any of these values can be changed as you copy the business object. Note that you only need to include clauses for the values that you want to change. If you do not make any changes, the values remain the same as the original business object with the new name you have assigned.

Modifying a Business Object

After a business object is added, you can change it with the Modify Businessobject statement. This statement lets you add or remove defining clauses or change some of their values.

```
modify businessobject OBJECTID [ ITEM {ITEM} ];
```

OBJECTID is the OID or Type Name Revision of the business object you want to modify.

ITEM is the type of modification you want to make. Each is specified in a Modify Businessobject clause, as listed in the following table. Note that you only need to specify fields to be modified.

Modify Businessobject Clause	Specifies that...
description VALUE	The current description, if any, is changed to the value entered.
image FILENAME	The image is changed to the new image in the file specified.
vault vault_NAME [update set]	The business object is moved from the current vault to the new vault specified here. See Changing an object's vault .
name NAME [revision REVISION]	The name and, optionally, the revision of the business object are changed to the name and revision specified here.
owner USER_NAME	The owner of the business object is changed to the owner specified here.
grant USER access ACCESS{,ACCESS} [signature true false] {grant USER access ACCESS{,ACCESS} [signature true false]} [key KEY]	The specified access to the object is granted to the user(s) specified. See Granting Access .
revoke all grantee grantor [USER]] {grantee grantor [USER]} [key KEY]	Access that has previously been granted to another user is revoked.
policy POLICY_NAME	The policy of the business object is changed to the policy specified here. Acquired signatures are retained if they match new signature requirements. The current state of the business object is retained if that state is a state of the new policy. If you change an object's policy you may also inadvertently change the store used by the object. In this case, newly checked in files will be kept in the new policy's store. If the object contained its own files before the policy was changed, they will stay in the old store, until/unless they are replaced during a subsequent checkin. However, the old store will still be used in the case where another object contains a reference to files in the object that now uses a different store. When the time comes for the reference to become an actual file (as when the file list changes between the 2 objects) the file copy is made in the same store the original file is located in.
move [format FORMATNAME] from BO_NAME FILES_TO_COPY;	The specified FILES_TO_COPY are removed from the from BO_NAME to the business object being modified. Refer to Moving Files below for more information.

Modify Businessobject Clause	Specifies that...
state STATE_NAME schedule DATE	The date associated with the beginning of a state is changed as specified here.
type TYPE_NAME	The type associated with the business object is changed as specified here. Attributes of the business object which are part of the new type definition are retained. Attributes which are not part of the new type definition are deleted. Any new attributes are added with their default value.
ATTRIBUTE_NAME VALUE	The attribute value associated with the business object is changed as specified here.
add history VALUE [comment VALUE]	A history Tag and a Comment can be specified with this clause. The Tag is the VALUE you enter after add history as a name for the custom history entry. A user/time/date/current stamp is automatically applied when this command executes. The comment is the VALUE (or text) you enter to describe the history item you are adding.
delete history [ITEM {ITEM}]	System administrators only can purge the history records of a business object. See Deleting History in Chapter 36 for details.

As you can see, each modification clause is related to the clauses and arguments that define the business object. When you modify a business object, you first name the object to be changed (by type, name, and revision) and then list the modification using the appropriate clauses.

Changing an object's vault

When an object's vault is changed, by default the following occurs behind the scenes:

- The original business object is cloned in the new vault with all business object data except the related set data
- The original business object is deleted from the “old” vault.

When a business object is deleted, it also gets removed from any sets to which it belongs. This includes both user-defined sets and sets defined internally. IconMail messages and the objects they contain are organized as members of an internal set. So when the object's vault is changed, it is not only removed from its sets, but it is also removed from all IconMail messages that include it. In many cases the messages alone, without the objects, are meaningless. To address this issue, you can fix sets at the time the change vault is performed (in MQL only). For example:

```
modify bus Assembly R123 A vault Engineering update set;
```

The additional functionality may affect the performance of the change vault operation if the object belongs to many sets and/or IconMails, which is why you must explicitly ask the system to do this.

Business administrators can execute the following MQL command to enable/disable this functionality as the default behavior for system-wide use:

```
set system changevault update set | on | off | ;
```

See [Controlling System-wide Settings](#) in Chapter 3 for more information.

Granting Access

Any person can grant accesses on an object to any other user as long as the person has “grant” access. The grantor is allowed to delegate all or a subset of his/her accesses on the current state.

The MQL command and ADK methods for granting business objects allow users to:

- grant an object to multiple users
- have more than one grantor for an object
- grant to any user (person/group/role/association)
- use a key to revoke access without specific grantor/grantee information

Custom ADK programs and MatrixOne Value Chain Portfolio applications can take advantage of the enhancements by using the MQL and ADK methods. However, desktop eMatrix and eMatrixApplet user interfaces do not support these changes.

The eMatrix history feature records one entry for each grant with the corresponding grantor, grantee, granteeaccess, and granteesignature. Previously, when there was a new access delegation, history recorded the new grantee and the previous grantee. However, with multiple grants, the new grantee is recorded but there is no historical record of the previous grantee.

For details on accesses, see [User Access](#) in Chapter 10.

Granting access to multiple users

You can use one `modify bus` command to grant an object to multiple grantees with the same or different accesses.

- To grant an object to multiple grantees, each with different accesses use:

```
modify bus OBJECTID grant USER access ACCESS{,ACCESS} [signature true|false] {grant  
USER access ACCESS{,ACCESS} [signature true|false]};
```

where:

`OBJECTID` is the OID or Type Name Revision of the object you want to modify.

`USER` is any person, group, role or association;

`ACCESS` is any of the different accesses that need to be provided to the grantee. (For details, see the [Accesses](#) in Chapter 10.) At least one `ACCESS` must be specified, or none are given. The `!` can be used with any access in conjunction with the keyword `all` to provide all privileges the delegator has, except those specified. For example:

```
modify bus Assembly SA-4356 A grant Engineering access all, !checkin;
```

This command will provide Engineering with all accesses except checkin. However, the Engineering group will only be able to perform operations for which the grantor has access (with the exception of checkin). “All” is evaluated just like other access items, and exceptions to “all” must be delimited with a comma.

The optional `signature` clause is used to provide the same access on signatures for the business object that the Grantor has. The default is `false`.

For example:

```
mod bus Part Sprocket 3 grant Jo access read signature false  
grant Jackie access read,modify signature true  
grant Jules access read,modify,checkin,checkout;
```

The above mod bus grants access to the object to three grantees: Jo, Jackie, and Jules. User Jo has only read access and cannot sign for the grantor. User Jackie has read and modify access and can sign for the grantor. User Jules has read, modify, checkin, checkout and cannot sign for the grantor.

- To grant an object to more than one grantee with the same accesses and signature privilege use:

```
mod bus TYPE NAME REVISION grant USER{,USER} access ACCESS{,ACCESS} [signature true|false];
```

For example:

```
modify bus Part Sprocket 3 grant Jo,Jackie,Jules access read,modify signature false;
```

Subsequent modify bus statements with the grant clause will overwrite, (not add to) the granted accesses and users *only* if the grantor and grantee match an existing grantor/grantee pair on the business object. If either the grantor or grantee of the pair does not match an existing pair, an additional set of accesses will be granted for the object.

Granting access using keys

Profile management tasks in the Value Chain Portfolio/ADK applications typically perform all grants from the same grantor, which makes identifying/modifying individual grants very difficult for the programmer. When granting business objects via MQL (or MQLCommand), it is possible to specify an identifier, or key, for each individual grant. Grants can then be revoked by referencing this key with or without the grantor/grantee information. To specify a key for a grant, use the following:

```
modify bus TYPE NAME REV grant USER access MASK key KEY;
```

For example:

```
modify bus Assembly 123 0 grant stacy access all key Buyer;
```

Revoking Access

You can use several approaches to revoke granted access to a business object:

- If you have Revoke access, you can remove all granted access on a business object with:

```
mod bus TYPE NAME REVISION revoke all;
```

For example:

```
mod bus Part Sprocket 3 revoke all;
```

- You can revoke all accesses granted by a specific grantor with:

```
mod bus TYPE NAME REVISION revoke grantor USER;
```

For example:

```
mod bus Part Sprocket 3 revoke grantor Jo;
```

The above command removes all grants made by grantor Jo. This command completes successfully if the context user is Jo, or if the context user has Revoke access; otherwise, an error is generated.

- You can revoke all accesses granted to a specific grantee with:

```
mod bus TYPE NAME REVISION revoke grantee USER;
```

For example:

```
mod bus Part Sprocket 3 revoke grantee Jackie;
```

This will revoke all grants where Jackie is the grantee. This command executes successfully if the current context is Jackie or if the current context has Revoke access; otherwise, an error is generated.

- You can specify both grantee and grantor to revoke the grant made between them with:

```
mod bus TYPE NAME REVISION revoke grantor USER grantee USER;
```

For example:

```
mod bus Part Sprocket 3 revoke grantor Jo grantee Jackie;
```

This will revoke the grant between grantor Jo and grantee Jackie. If the context user is Jo, or if the context user has Revoke access, this command will succeed; otherwise, an error will be generated.

- To revoke the grant without specific grantor/grantee information, you can specify an identifier, or key, defined when the grant was made:

```
mod bus TYPE NAME REV revoke key KEY;
```

In addition, the key is selectable as follows:

```
print bus|set TYPE NAME REV select grantkey;
expand bus|set TYPE NAME REV select grantkey;
```

The following example demonstrates the use of keys to grant and revoke access, and includes MQL print bus output on the object:

```
MQL< >mod bus Assembly 123 0 grant bill access read,checkin,show key Supplier;
MQL< >mod bus Assembly 123 0 grant bill access read,checkin,checkout,show key
SupplierEngineer;
MQL< >mod bus Assembly 123 0 grant stacy access all key Buyer;
MQL< >print bus Assembly 123 0 select grant.*;
business object Assembly 123 0
    grant[creator,bill].grantor = creator
    grant[creator,bill].grantor = creator
    grant[creator,stacy].grantor = creator
    grant[creator,bill].grantee = bill
    grant[creator,bill].grantee = bill
    grant[creator,stacy].grantee = stacy
    grant[creator,bill].granteeaccess = read,checkin,show
    grant[creator,bill].granteeaccess = read,checkin,checkout,show
    grant[creator,stacy].granteeaccess = all
    grant[creator,bill].granteesignature = FALSE
    grant[creator,bill].granteesignature = FALSE
    grant[creator,stacy].granteesignature = FALSE
    grant[creator,bill].grantkey = Supplier
    grant[creator,bill].grantkey = SupplierEngineer
    grant[creator,stacy].grantkey = Buyer
```

Since a unique name (key) has been provided for each grant, the grants can be revoked using this key, rather than relying on the grantor, grantee pair, which may not be -- and in this example is not -- unique. For example:

```
MQL< >mod bus Assembly 123 0 revoke grant key SupplierEngineer;
```

This command removes the access granted to Bill as a SupplierEngineer, but it will leave the access granted to Bill as a Supplier. The print command output confirms this:

```

MQL< >print bus Assembly 123 0 select grant.*;
business object  Assembly 123 0
grant[creator,bill].grantor = creator
grant[creator,stacy].grantor = creator
grant[creator,bill].grantee = bill
grant[creator,stacy].grantee = stacy
grant[creator,bill].granteeaccess = read,checkin,show
grant[creator,stacy].granteeaccess = all
grant[creator,bill].granteesignature = FALSE
grant[creator,stacy].granteesignature = FALSE
grant[creator,bill].grantkey = Supplier
grant[creator,stacy].grantkey = Buyer

```

Be aware that Revoke access is a very powerful access. By giving a user Revoke access, you are saying that this user can revoke anyone's grants.

Moving Files

You can move files from one object to another (or another format of the same object) using the `modify bus` command, provided you have checkout access to the from object and checkin access to the to object. Moving files is equivalent to doing a checkout, delete file and then checkin to a new format or object. However, the files are not actually moved at all (the files stay in the same store - same physical machine location). The command simply changes the metadata so that the files no longer appear to belong to the original object but now belong to the new object.

```
modify businessobject BO_NAME move [format FORMATNAME] from BO_NAME FILE_TO_COPY;
```

where `FILES_TO_COPY` is one of:

```

format FORMATNAME
[format FORMATNAME] file FILENAME{,FILENAME}
all

```

Files are copied (appended) from the object specified in the from clause (the “from object”) to the object being modified (the “to object”). If a format is specified just after the keyword `move`, the files will be moved to that format. Otherwise, each file will keep its existing format as long as that format is available in the to object. If not, it will use the default format of the to object.

If only a format is given for the from object (part of `FILES_TO_COPY`), all files of that format are moved. If no format is specified, then the default format is assumed. If any of the specified files is not found with the specified or assumed format, an error is issued and no action is taken.

No triggers are fired when files are moved in this manner.

Creating a Revision of an Existing Business Object

The ability to create revisions can be granted or denied depending on the object's state and the session context. For example, let's assume you are an editor working on a cookbook. A cookbook is composed of Recipe objects. Recipe objects are created by the Chef who writes the recipe, perhaps in the description field of the object. He then promotes the Recipe to the "Test" state, where he makes the dish and tastes it. At this point, he either approves it and sends it to the next state (perhaps "Submitted for Cookbook"), reassigning ownership to you (the editor), or he may want to revise it, incorporating different ingredients or varying amounts. Therefore, the "Test" state would have to allow revisions by the owner. The Recipe object could then be revised, the new revision starting in the first state of the lifecycle. Once the Recipe is approved, revisions should not be allowed; so, the "Submitted for Cookbook" state would not allow revisions.

To revise a business object, use the Revise Businessobject statement:

```
revise businessobject OBJECTID [to REVISION_NAME];
```

OBJECTID is the OID or Type Name Revision of the object you want to revise.

REVISION_NAME is the revision designator to be assigned.

In order to maintain revision history, the new object must be created with the revised business object statement even though it is possible to create what looks like a new revision by manually assigning a revision designator with the Add or Copy Businessobject statement. The table below shows the differences between using the `revise businessobject` statement and the `copy businessobject` statement.

Differences/Similarities between Clones and Revisions		
Property	Copy or Clone	Revision
Attributes	Values are initially the same. Dialog appears upon creation for modification.	Values are initially the same. Dialog appears upon creation for modification.
Files	Files are copied to the Clone upon creation	Files are referenced from the original until it is necessary to copy them.
Connections to other objects	Depends on the Revision Rules (Float, Replicate, None) set by the Business Administrator.	Depends on the Revision Rules (Float, Replicate, None) set by the Business Administrator.
Connection to Original	No implicit connection.	Implicit connection can be viewed in Revision chain.
History	Nothing in history show that it evolved from its original.	The create entry shows object from which it was been revised and original shows that it was revised.

Not all business objects can be revised. If revisions are allowed, the Sequence clause of the Policy definition specifies the scheme for labeling revisions. This scheme can include

letters, numbers, or enumerated values. Therefore you can have revisions with labels such as AA, 31, or “1st Rev.”

If the REVISION_NAME is omitted, eMatrix automatically assigns a designator based on the next value in the sequence specified in the object’s policy. If there is no sequence defined, an error message results.

If there is no defined revision sequence or if you decide to skip one or more of a sequence’s designators, you can manually specify the desired designator as the REVISION_NAME.

For example, assume you entered the following statement:

```
revise businessobject "Shipping Form" "Lyon's Order" 5;
```

Since no revision designator is given, the new business object will be called the following:

```
"Shipping Form" "Lyon's Order" 6
```

If you want to skip over the next revision numbers and assign a different number, you can do so as:

```
revise businessobject "Shipping Form" "Lyon's Order" 5 to 10;
```

This statement will produce a new business object labeled:

```
"Shipping Form" "Lyon's Order" 10
```

Now the revised business object has a revision designator equal to ten.

Note that you cannot use a designator that has already been used. If you do, an error message will result.

Adding an Object to a Revision Sequence

You can use MQL to add an object to the end of an existing revision sequence, as long as the object is not already a member of another revision sequence. Attempts to add an object in the middle of an existing revision chain, or to add an object that is already part of a revision sequence will cause the MQL command to error.

Creating new revisions in eMatrix has several side affects. When appending to a revision sequence with the MQL command, these behaviors are handled as described below:

- **Float/Replicate rules.** Ordinarily the float/replicate rules for relationships cause relationships to be moved/copied to new revisions. These rules are ignored; no relationships are added or removed to/from the inserted object, nor are any relationships added or removed to/from any of the previously existing members of the target sequence.
- **File Inheritance.** Ordinarily, a new revision of an existing object inherits all files checked into the original object. When using this interface, if the appended object already has checked in files, it does not inherit any files from the revision sequence. If the appended object has no files checked in, the behavior is controlled by the file keyword in MQL.
- **Triggers.** No triggers will fire.
- **History.** Revision history records will be recorded on both the new object and its previous revision (that is, both objects that are specified in the MQL command).

To add an object to an existing revision sequence, use the Revise Businessobject statement:

```
revise businessobject OBJECTID bus NEWOBJECTID [[not]file];
```

OBJECTID is the OID or Type Name Revision of the last object in a revision chain.

NEWOBJECTID is the OID or Type Name Revision of an existing object that is not already part of any revision sequence.

The optional `file` keyword applies only if the new object has no files checked in, in which case it specifies whether files should or should not be inherited from the revision sequence to which it is being added. The default is `not file`, so no files are inherited. This flag is ignored if NEWOBJECTID contains files.

Deleting a Business Object and Its Files

You can remove an entire business object (including all checked in files) or you can remove single files from the business object with the Delete Businessobject statement:

```
delete businessobject OBJECTID [[format FORMAT_NAME] file FILENAME{,FILENAME}];  
Or  
delete businessobject OBJECTID [[format FORMAT_NAME] file all];
```

OBJECTID is the OID or Type Name Revision of the business object to be deleted or from which files should be deleted.

FORMAT_NAME is the name of the format of the files to be removed from the business object. If none is specified the default format is assumed.

FILENAME indicates a checked in file to be removed from the business object. You can use the All argument (rather than FILENAME) to remove all checked in files for a given format.

When this statement is processed, eMatrix searches the list of business objects. If the named business object is found, any connections it has are first removed, (so that history is updated on the object on the other ends) and then that object is deleted along with any associated files. If the name is not found, an error message results.

For example:

```
delete businessobject Person "Cheryl Davis" 0;
```

After this statement is processed, the business object is deleted and any relationships with that object are dissolved. You will receive the MQL prompt for another statement.

To remove a checked in file of format ASCII text, enter the following statement:

```
delete businessobject Assembly "Telephone Model 324" AD  
format "ASCII Text"  
file assemble324.doc;
```

Deletion of files can occur automatically, as well, during a checkin/replace, removal of a business object, move file, or synchronization of file stores. In a replicated environment, the delete function removes all copies of each file at each location. For more information about checked-in files, refer to [Checking Files In and Out of a Business Object](#) in Chapter 35.

35

Working with Business Objects

Making Connections Between Business Objects.....	724
Connect Businessobject Statement.....	724
Disconnect Businessobject Statement	725
Modify Connection Statement.....	726
Displaying and Searching Business Object Connections.....	728
From Clause	729
To Clause	729
Relationship Clause.....	730
Type Clause.....	730
Filter and Activefilter Clauses	731
Recurse Clause	731
Set Clause	732
Structure Clause.....	732
SELECT_BO and SELECT_REL Clauses	733
Dump Clause and Output Clause	734
Tcl Clause.....	736
Recordseparator Clause.....	736
Terse Clause	736
Limit Clause	736
Working with Saved Structures.....	737
Checking Files In and Out of a Business Object.....	738
Checking In Files	738
Checking Out Files	741
Moving Files.....	743
Locking and Unlocking a Business Object.....	744
Locking a Business Object	744
Unlocking a Business Object.....	745
Modifying the State of a Business Object.....	746
Approve Businessobject Statement.....	746
Ignore Businessobject Statement.....	747
Reject Businessobject Statement.....	747
Unsign Signature	748
Disable Businessobject Statement	748
Enable Businessobject Statement.....	748
Override Businessobject Statement	749
Promote Businessobject Statement.....	750
Demote Businessobject Statement.....	750

Making Connections Between Business Objects

As described in *Overview of Relationships* in Chapter 14, relationship types are created which can be used to link business objects. A *relationship type* is specified when making a *connection* between two business objects. One business object is labeled as the TO end and one is labeled as the FROM end. When the objects are equivalent, it does not matter which object is assigned to which end. However, in hierarchical relationships, it does matter. eMatrix will use the TO and FROM labels to determine the direction of the relationship.

The direction you select makes a difference when you examine or dissolve connections. When you examine an object's connections, you can specify whether or not you want to see objects that lead to or away from the chosen object. When you disconnect objects, you must know which object belongs where. Therefore, you should always refer to the relationship definition when working with connections.

In the sections that follow, you will learn more about the statements that connect and disconnect business objects.

Connect Businessobject Statement

The Connect Businessobject statement links one business object to another. For example, you could have a business object that contains information about a particular course. That information might include the course content, schedule date, instructor, student list, cost, and so on. As each student enrolls in the course, you might create a business object for that student. This object might include the student's background, experience, and personal information.

After a student record object is created, it stands alone with no relationships attached to it. However, if you view the course object, you might want to see the student objects associated with it. Also, if you view the student object, you might want to see the course objects associated with that person.

Use the Connect Businessobject statement to establish the relationship between the business object containing the student record and the object containing the course information:

```
connect businessobject OBJECTID relationship NAME [to|from] OBJECTID [ATTRIBUTE_NAME  
VALUE {ATTRIBUTE_NAME VALUE}];
```

OBJECTID is the OID or Type Name Revision of the business object. It can also include the in VAULTNAME clause, to narrow down the search.

NAME is the name of the relationship type to use to connect the two named business objects. If the relationship name is not found or defined, an error message will result.

ATTRIBUTE_NAME VALUE indicates attributes assigned to the relationship you are creating.

The Connect Businessobject statement has two forms: TO and FROM. The form you choose depends on the placement of the two objects you are connecting. You specify which business object will be associated with the TO end and which will be associated

with the FROM end. For example, to assign a student to a course, you might use the TO form of the Connect Businessobject statement:

```
connect businessobject Student "Cheryl Davis" Sophomore  
    relationship "Student/Course Relationship"  
        to Course "C Programming Course" 1;
```

When this statement is processed, it will establish a “Student/ Course Relationship” between the course object and the student object. Cheryl Davis is assigned to the FROM end and the C Programming course is assigned to the TO end. You can think of Cheryl as *leading to* the course object.

You also could think of the course as coming *from* the student object(s). In other words, you can define the same relationship using the FROM form of Connect Businessobject statement:

```
connect businessobject Course "C Programming Course" 1  
    relationship "Student/Course Relationship"  
        from Student "Cheryl Davis" Sophomore;
```

When this relationship is established, Cheryl’s object is again assigned to the FROM end and the course is assigned to the TO end.

If you are defining equivalent objects, either object could be defined as the TO end or the FROM end. However, in hierarchical relationships, direction is important. With these relationships, you should consult the relationship definition before creating the connection. Otherwise, you may have difficulty locating important objects when needed. If you assigned the wrong object to the connection end, you will have to dissolve the relationship and re-define it.

Disconnect Businessobject Statement

The Disconnect Businessobject statement dissolves a link that exists between one business object and another. For example, a student enrolled in a class might discover that other commitments make it impossible to attend. Since the student is withdrawing from the class, you no longer need a relationship that was established between that student and the class.

Use the Disconnect Businessobject statement to dissolve the relationship between the business object containing the student record and the object containing the course information:

```
disconnect businessobject OBJECTID relationship NAME [to|from] OBJECTID;
```

OBJECTID is the OID or Type Name Revision of the business object. It may also include the *in VAULTNAME* clause, to narrow down the search.

NAME is the name of the relationship type that exists between the two named business objects. If the relationship name is not found or defined, an error message will result.

The Disconnect Businessobject statement has two forms: TO and FROM. The form you choose depends on the placement of the two objects you are disconnecting. In our

example, the student was assigned to the FROM end and the course was assigned to the TO end. Therefore, to dissolve this relationship, you can write either of these statements:

```
disconnect businessobject Student "Cheryl Davis" Sophomore  
relationship "Student/Course Relationship"  
to Course "C Programming Course" 1;
```

```
disconnect businessobject Course "C Programming Course" 1  
relationship "Student/Course Relationship"  
from Student "Cheryl Davis" Sophomore;
```

In the first statement, the TO form is used. You are dissolving the relationship of student *to* the course. In the second statement, the FROM form is used. You are dissolving the relationship *from* the student.

Regardless of the form you use, once processed, the relationship that was established between the student and the course is removed.

The direction you select makes a difference when you are dissolving relationships. If you reversed the order of the two business objects, the names would not be found. For example, assume you entered this statement:

```
disconnect businessobject Student "Cheryl Davis" Sophomore  
relationship "Student/Course Relationship"  
from Course "C Programming Course" 1;
```

This statement assumes that the student is the TO end and the course is the FROM end. If eMatrix searched for this relationship, a match would not be found and an error message would result. You should check to be sure you know the correct direction before writing your Disconnect Businessobject statement. See [Displaying and Searching Business Object Connections](#) in the next section to learn how to search for relationships.

Modify Connection Statement

To change a Relationship type, the relationship instance can be specified by its defined name together with the objects on both ends of the connection as follows:

```
modify connection businessobject OBJECTID to|from businessobject OBJECTID type  
REL_NAME;
```

OBJECTID is the OID or Type Name Revision of the business object. It may also include the `in VAULTNAME` clause, to narrow down the search.

REL_NAME is the new Relationship type name.

For example:

```
modify connection bus Assembly 50463 A to Component 33457-4 G type "As Designed";
```

If the object is connected to another object by more than one instance of the specified relationship type, the above command will generate an error. To make modifications when that is the case, the relationship ID must be specified. To obtain the relationship IDs of all connections, use the `relationship ID` selectable item. For example:

```
expand bus Assembly 50402 B select relationship id;
```

The following is an example of the result, which could be saved to a file if an output clause was used in the above command:

```
1 Drawing from Drawing 50402 A  
    id = 19.24.5.16  
1 Process Plan from Process Plan 50402-1 C  
    id = 19.26.6.6
```

The returned list of IDs can then be used to make changes to any of the relationships, using the following syntax:

```
modify connection ID type REL_NAME;
```

For example:

```
modify connection 19.10.6.13 type Documents;
```

Changing Relationship Ends

It is possible to specify a new object to replace the existing object on the end of a relationship. This is comparable to using the drop connect feature in eMatrix, and dropping an object on top of a relationship arrow (to replace the child object) in the Navigator window. The operation actually performs a disconnect and then a connect action, and so relies on those accesses to determine if the user can perform the action. The user must also have modify access on the relationship. Use the keywords `to` and `from` on the `modify connection` commands as follows:

```
modify connection ID from|to OBJECTID;
```

Or

```
modify connection bus OBJECTID from|to OBJECTID relationship  
NAME from|to NEW_OBJECTID;
```

`OBJECTID` is the `OID` or Type Name Revision of the business object.

To change the direction of the relationship, use both a ‘`to`’ and a ‘`from`’ `OBJECTID` clause. For example, assume that Assembly 50402 B is currently on the from end of the relationship. The following command would reverse the direction:

```
modify connection 62104.18481.14681.40078 to Assembly 50402 B  
from 62104.18481.13184.46733;
```

Notice that any combination of IDs and full specification by type name and revision (or relationship name) are allowed. When using type, name and revision you may also include in `VAULTNAME`.

Deleting History

System administrators only can purge the history records of a connection via MQL. For details, see [Deleting History](#) in Chapter 36.

Displaying and Searching Business Object Connections

The Expand Businessobject statement enables you to view connected objects. Depending on the form of this statement that you use, you can list the objects that lead to an object, from it, meet selected criteria, or are of a specific type. Then you can place the list of objects into/onto a set, or output the information to a file.

The Expand Businessobject statement also allows you to select properties of either connected business objects or the relationship which links them. To expand a business object use the following syntax:

```
expand businessobject OBJECTID {EXPAND_ITEM} ;
```

OBJECTID is the OID or Type Name Revision of the business object.

EXPAND_ITEM can be any of the following

```
from | to  
relationship PATTERN  
type PATTERN  
filter PATTERN  
activefilters [reversefilters]  
recurse [to | N | ]  
| all |  
| into | set NAME  
| onto |  
structure NAME  
SELECT_BO  
SELECT_REL  
DUMP [RECORDSEP]  
tcl  
output FILENAME  
terse  
limit N
```

The Expand Businessobject statement is similar to using the Star or Indented browser in eMatrix Navigator. Use of the clauses that specify criteria is like applying a filter in one of the relationship browsers. In addition to a list of connected business objects that meet the specified criteria, the Expand Businessobject statement provides information about the connections. The level number of the expansion and the relationship type of the connection are provided along with the business object name, type, and revision.

The following example shows basic use of the Expand Businessobject statement. It tells us that there is one drawing attached to a component (with the relationship of “Drawing to”),

a comment (with the relationship of “Comment from”) and a markup (with the relationship of “Markup from”). The number “1” on each line indicates the expansion level.

```
expand businessobject Drawing 726592 A;
  1 Drawing to Component 726592 A
  1 Comment from Comment 012528 1
  1 Markup from Markup 002670 1
```

When using the `Expand Businessobject` statement you must provide a business object specification as your starting point. Depending on the keyword (and values) that follow, you will *expand* in one direction only or in both. In addition, you can use the `recurse` argument to indicate that you also want to expand the business objects connected/related to the initially specified business object.

The expansion happens one level at a time, and any relationships/business objects that do not satisfy the where clause are removed from the list and those expansion paths are abandoned. Consider the following example:

Assembly -> Part -> Doc

```
expand bus Assembly from recurse 2 where 'type == Doc' will
return nothing, since the first level expansion finds Part, which is not of type Doc, so the
second expansion is not performed at all.
```

Each clause is described in the sections that follow.

From Clause

When you use the `From` clause, you expand away from the starting object. This gives you all of the related objects defined as the TO ends in a relationship.

Think of the starting object as the *tail* of the arrow—you are looking for all of the arrow *heads*.

For example, assume you have an object that contains a component used within larger assemblies. If you have the name of the component, you might want to know which objects (larger assemblies) it goes into:

```
expand businessobject Component "Mini Numeric Keypad" C from;
```

This might produce a list of objects that could include a calculator, telephone, and VCR. Since no other criteria is specified, this statement gives you ALL objects that occupy the TO end of a relationship definition.

To Clause

When the `To` clause is used, the named business object is used as the starting point. However, it is assumed that the object is defined as the TO end of the relationship definition. Therefore, you are looking for all objects that lead to the named object—the objects defined as the FROM ends in a relationship definition.

Using the `To` clause can be useful to work backward. For example, you may want to determine components that make up a particular subassembly. If you know the name of the subassembly object, you can do this by writing a statement similar to:

```
expand businessobject Component "Mini Numeric Keypad" C to;
```

This might produce a list of objects that contain buttons, plastic housings, and printed circuit boards. All related objects defined as the FROM connection end are listed. This may include additional objects that you don't need. To help reduce the number of objects

listed and to allow you to look in both directions, you can use the Relationship or Type clauses of the Expand Businessobject statement.

Relationship Clause

The Relationship clause of the Expand statement returns all objects connected to the starting object with a specific relationship. The command can be made to be more particular by specifying the direction of the relationship and/or the types of objects to return.

The Relationship clause is useful when you are working with an object that may use multiple relationship types. If the starting object can connect with only one relationship, this clause has the effect of listing all of the connection ends used by the starting object. It does not matter if the end is defined as a TO end or a FROM end. Only the relationship type is important.

For example, assume you have an object that contains a drawing. This drawing may use a number of relationships such as a User Manual Relationship, Design Relationship, Marketing Relationship, and Drawing Storage Relationship. You may want to examine all objects that use a particular relationship type. For example, you might want to learn about all objects that have used the drawing for marketing. To do this, you might enter a statement similar to:

```
expand businessobject Drawing "Mona Lisa" 1  
relationship "Marketing Relationship";
```

This statement lists all related objects that have the Marketing Relationship type. It searches through all connections that use the Marketing Relationship definition for the named business object. If it finds the object, MQL displays the other connection end and identifies the end's directional relationship. It does not matter if the related objects are defined as the FROM end or the TO end of the relationship. Only the relationship type is important.

Type Clause

The Type clause of the Expand Businessobject statement displays all related objects of a specific object type. This clause is useful when you are working with an object that may be connected to multiple object types. (If the starting object can connect with only one object type, this form is similar to the Relationship form using a wild character pattern.) For example, assume you have an object that contains a training course. To this object you might have related objects that are of type Evaluation, Student, Materials, and Locations. You may want to examine all the Evaluation type objects to trace the course's progress in meeting student needs. You might enter a statement similar to:

```
expand businessobject Course "Money & Marketing" 3 type Evaluation;
```

This statement lists all related objects that have the Evaluation type. Those objects might belong to multiple relationship types (such as Professional Evaluation Relationship or Student Evaluation Relationship). It does not matter if the related objects are defined as the FROM end or the TO end of the relationship. Only the object type is important to locate and display the output objects.

If the specified type has sub-types, these are also listed. For example:

```
expand bus Assembly 12345 1 type Part;
```

might return all Component and Assembly, as well as Part objects.

Filter and Activefilter Clauses

The filter clause of the Expand Businessobject command allows you to specify a named filter(s) to be used for the expansion. You can use wildcard characters or an exact name. For example:

```
expand bus Assembly 12345 1 filter Part;
```

In addition, you can use the activefilter clause to indicate that you want to use all filters that are enabled in the current user's context. For example:

```
expand bus Assembly 12345 1 activefilter;
```

Recurse Clause

Once you have a list of related objects, you may also want to expand these objects. This would be like using the There and Back feature in the Star browser of eMatrix Navigator. The Recurse clause of the Expand Businessobject statement expands through multiple levels of hierarchy by applying the Expand statement to each business object found:

```
recurse to [N|all]
```

N is any number indicating the number of levels that you want to expand.

all indicates that you want to expand all levels until all related business objects are found.

It is recommended that you specify a recursion level with the Recurse clause. Recurse to all may be time-consuming.

This example shows use of the Expand Businessobject statement with the Recurse (to all) clause:

```
MQL<40> expand businessobject Drawing 726592 A recurse to all;
```

```
1 Drawing from Component 726592 A
2 As Designed from Assembly 726509 A
3 As Designed to Component 726593 A
4 Drawing from Drawing 726593 a
5 Comment from Comment 123528 1
6 Comment to Drawing 726592 A
7 Markup from Markup 002670 1
3 As Designed to Component 726594 A
4 Drawing from Drawing 726594 A
5 Markup from Markup 002590 1
3 As Designed to Component 726595 A
4 Drawing from Drawing 726595 A
5 Markup from Markup 002733 1
4 Note from Note 014258 1
5 Note to Component 726591 B
6 Note from Note 012499 1
6 Photo from Photo 017012 1
6 Photo from Photo 117012 1
6 Drawing from Drawing 726591 C
7 Markup from Markup 002715 1
6 As Designed from Assembly 726596 B
7 As Designed to Component R0056-48 A
7 As Designed from Assembly 726590 A
8 As Designed to Component R0045-62 B
8 As Designed to Component 23348S C
```

```
8 As Designed to Component W2236-8S A
8 Comment from Comment 013070 1
8 Drawing from Drawing 726590 B
8 Layout from Layout 100265 F
8 Analysis from Stress Analysis 100345 D
9 Specification from Specification 2278 C
10 Specification Change from Spec. Change Notice 00247 1
10 Specification Change from Spec. Change Notice 00252 1
7 Drawing from Drawing 726596 B
```

Set Clause

Once you have a list of related objects, what do you do with them? In some cases, you will simply search for a particular object and will not need to reference the output object again. In other cases, you may want to capture the output business objects and save them for another time. That is the purpose of the Set clause of the Expand Businessobject statement.

When the Set clause is included within the Expand Businessobject statement, the related objects are placed within a set and assigned a set name. This set may already contain values or it may be a new set created for the purpose of storing the output.

When it is an existing set, the previous values are either replaced or appended, depending on the keyword you use. If the Set clause begins with the keyword `INTO`, the existing set contents are discarded and only the current output is saved. If the Set clause begins with the keyword `ONTO`, the new output is added onto the existing set contents. (This is the same as when working with queries, described in [Query Overview](#) in Chapter 38.)

The Set clause is optional. It can be used in conjunction with the other Expand Businessobject clauses according to the following guidelines:

- You can specify a set (with the Set clause) *OR* select properties for output (with the Select clause) but not both in the same command.
- You can specify that the results are saved in a set *OR* request that output be saved in a file (with dump/output clauses), but not both in the same command.

Structure Clause

If you expand an object using the `filter`, `from`, `to`, `relationship`, and/or `type` clauses of the Expand Businessobject command, you have defined a *structure* that is not necessarily easy to define again. If you want to reuse this structure you can save it to your workspace using the structure clause. For example:

```
expand bus Assembly 12345 1 from relationship "As Designed" filter Part structure
"Assigned Parts";
```

A structure stores all the information—including objects, relationships, and relative level number — necessary to recreate the printed output. The structure clause is optional. It can be used in conjunction with any of the other Expand Businessobject clauses, including select, output, and set.

If an object has been disconnected or deleted, it is no longer listed as part of the structure. If the deleted or disconnected object was expanded as part of the structure, its “child” objects would also be removed from the saved structure.

On the other hand, if other objects are connected after the structure is saved, they will not automatically appear in the output of the print structure command.

Also, if a business object (and its relationships) are moved to another vault, the structure will usually still remember them. However, if an object is moved to a vault in which no other object in the structure currently resides, the object will not be found. If this object is the “top” object in the structure (that is, the object that was expanded to create the structure), an error will occur if one tries to do anything with the structure. If the “lost” object is a child in the structure, it and all objects under it will not be listed as part of the structure.

Another expand statement would need to be executed with the structure clause to update the structure, in any of these cases.

Refer to [Working with Saved Structures](#).

SELECT_BO and SELECT_REL Clauses

The Expand Businessobject statement provides a means of displaying information about the connected business objects. Information contained in the connected business object can be selected as well as data on the relationship that connects the objects. It works in a similar manner on the expand statement as it does on the print statement. The differences are:

- a list of the selected values, one for each object or relationship that meets the criteria, is presented.
- the keyword `businessobject` or `relationship` must be used with the select clause before the requested items so that eMatrix knows from where the information is to come.

For example:

```
expand bus Assembly "MTC 1234" C select bus description select relationship id;
```

The system attempts to produce output for each select clause input, even if the object does not have a value for it. If this is the case, an empty field is output. For information on the `select` clause, see [Select Statements](#) in Chapter 34. Also refer to [Appendix A, Select Expressions](#) in the *eMatrix Navigator Guide*.

When you use the `select` clause to expand on both business objects and relationships, business object selectables are always listed before relationship selectables in the output, even if they appear in the opposite order in the `select` list.

Using a where QUERY_EXPR clause

When expanding business objects, you can use where clauses as another means of specifying which objects to list. When using this technique, you must always insert `select bus` or `select rel` before the where clause. This tells the system whether to apply the where clause to the relationships it finds or to the objects on the other end of those relationships. It does not, however, select and print out any information about the connected objects or expanded relationships (unless another `SELECT_BO` or `SELECT_REL` clause is included).

Applying the where clause to business objects

```
mql expand bus Type_one Object_1 Rev_1
    select bus
    where '(type == "Type two") && (1 == 1)' dump
```

Applying the where clause to relationships

Note that since the where clause works off the rel, you must use ‘to.type’.

```
mql expand bus Type_one Object_1 Rev_1  
    select rel  
    where ' (to.type == "Type two") && (1 == 1) ' dump
```

Alternatively, you can let expand bus check the object type for you (see Rules of thumb below):

```
mql expand bus Type_one Object_1 Rev_1  
    type "Type two"  
    select rel  
    where ' (1 == 1) ' dump
```

Note that the first command above is a bit more general, ignoring the direction of the relationship, whereas the second will only find Type two’s at the end of relationships pointing *from* Object_1 *to* the Type 2 object.

Reworking the first syntax for more specificity:

If you only want objects connected to Object_1 by the relationship Rel, add ‘rel Rel’:

```
mql expand bus Type_one Object_1 Rev_1  
    rel Rel  
    select bus  
    where ' (type == "Type two") && (1 == 1) ' dump
```

And if you only want to look at the relationship Rel coming *from* Object_1, add ‘from rel Rel’. This is equivalent to the REL syntax above:

```
mql expand bus Type_one Object_1 Rev_1  
    from rel Rel  
    select bus  
    where ' (type == "Type two") && (1 == 1) ' dump
```

Rules-of-thumb for where clauses:

- Always enclose the entire where clause in single quotes and a space.
- Always surround operators with spaces.
- In expand bus, if you can safely be specific about the relationship type and direction to be expanded, always use the syntax:

```
expand bus T N R <direction> rel <relnameList> select <rel-or-bus> where...  
because it is quicker. It will return only the correctly named and directed relationships and filter those relationships (or the business objects on the other ends for select bus) through the where clause.
```

- If you also know which Types you are looking for, use the type modifier for expand bus rather than relegate it to the where clause:

```
expand bus T N R from rel Rel type "Type two"  
    select rel where ' <other qualifiers on the rel> '  
    select bus where ' <other qualifiers on the TO obj's> '
```

Dump Clause and Output Clause

You can use the Set clause (described above) or the Dump/Output clause, but not both clauses.

You can specify a general output format for listing the expanded information to a file and for output. This is done with the Dump clause:

```
[dump "SEPARATOR_STR"] [output FILENAME]
```

SEPARATOR_STRING is a character or character string that will appear between the field values. It can be a tab, comma, semicolon, carriage return, and so on. If you do not specify a separator string value, a space is used. If a tab or carriage return is used as a separator, double quotes (" ") are required.

FILENAME identifies a file where the print output is to be stored.

The Dump clause specifies that you do not want to print the leading field name (a space) and that you want to separate the field names with the separator string you provide.

Separator strings can make the output more readable. If many of the business objects have similar field values, using tabs as separators will make the values appear in columns. The Output clause prints the expanded information to a file that you specify (FILENAME).

This example shows use of the Expand Businessobject statement with the Dump clause:

```
MQL<42> expand businessobject Drawing 726592 A recurse to all dump ":::";
```

```
1::Drawing::from::Component 726592 A
2::As Designed::from::Assembly 726590 A
3::As Designed::to::Component 726590 A
4::Drawing::from::Drawing 726593 A
5::Comment::from::Drawing 012528 1
6::Comment::to::Drawing 725492 A
7::Markup::from::Markup 002670 1
3::As Designed::to::Component 726594 A
4::Drawing::from::Drawing 726594 A
5::Markup::from::Markup 002590 1
3::As Designed::to::Component 726595 A
4::Drawing::from::Drawing 726595 A
5::Markup::from::Markup 002733 1
4::Note::from::Note 014258 1
5::Note::to::Component 726591 B
6::Note::from::Note 012499 1
6::Photo::from::Photo 017012 1
6::Photo::from::Photo 117012 1
6::Drawing::from::Drawing 726591 C
7::Markup::from::Markup 002715 1
6::As Designed::from::Assembly 726596 B
7::As Designed::to::Component R0056-48 A
7::As Designed::from::Assembly 726590 A
8::As Designed::to::Component R0045-62 B
8::As Designed::to::Component 23348S C
8::As Designed::to::Component W2236-8S A
8::Comment::from::Comment 013070 1
8::Drawing::from::Drawing 726590 B
8::Layout::from::Layout 100265 F
8::Analysis::from::Stress Analysis 100345 D
9::Specification::from::Specification 2278 C
10::Specification Change::from::Spec. Change Notice 00247 1
10::Specification Change::from::Spec. Change Notice 00252 1
7::Drawing::from::Drawing 726596 B
```

Tcl Clause

Use the Tcl clause after the dump clause, if used, and before the output clause to return the results in Tcl list format. This facilitates the parsing of output from MQL select commands within Tcl code since the built-in list handling features of Tcl are used directly on the results. For more information, see [Tcl Format Select Output](#) in Chapter 1.

Recordseparator Clause

The Recordseparator clause of the Expand Businessobject statement allows you to define which character or character string you want to appear between the dumped output of each object when the expand command requests information about multiple objects.

```
recordseparator [ "SEPARATOR_STR" ]
```

SEPARATOR_STR is a character or character string that replaces the end-of-line (\n) ordinarily inserted at the end of each object's dumped output. It can be a tab, comma, semicolon, etc. If tabs are used, they must be enclosed in double quotes ("").

Terse Clause

You can specify the terse clause so that object IDs are returned instead of type, name, and revision. This is done with the Terse clause. For example, the following statement returns a list of object IDs for objects connected to the specified Part:

```
expand businessobject Part "35735" A terse;
```

Limit Clause

Since you may be accessing very large databases, you should consider *limiting* the number of objects to display. Use the Limit clause to define a value for the maximum number of objects to include in the expansion. For example, to limit the expansion to 25 objects, you could use a statement similar to the following:

```
expand businessobject Part "35735" A limit 25;
```

Working with Saved Structures

Once you have saved a structure using the [Structure Clause](#) of the Expand Businessobject command, you can list, print, delete, and save it to another user's workspace using the following commands.

```
list structure;  
  
print structure NAME;  
  
delete structure NAME;  
  
copy structure SRC_NAME DST_NAME [fromuser USER_NAME] [touser  
USER_NAME] [overwrite];
```

The `print structure` command displays the results in the same manner as `expand bus` does. For example, after executing the above statement (which outputs the data to the MQL window, as well as saving it as a structure), you could execute the following to generate the output again:

```
print structure "Assigned Parts";
```

Within the `print structure` command you can also use select clauses on either the business object or the relationship as well as use the output/dump or terse clauses.

The `copy structure` command lets you copy structures to and from any kind of user. Including `overwrite` will replace the copied structure with any structure of the same name that was in the touser's workspace.

If an object has been disconnected or deleted, it is no longer listed as part of the structure. On the other hand, if other objects were connected since the structure was saved, they would not automatically appear in the output of the `print structure` command. Another `expand` statement would need to be executed with the `structure` clause to update the structure.

Checking Files In and Out of a Business Object

An individual file or an entire directory including all files and subdirectories can be checked in/out of a business object.

Checking In Files

A business object does not need to have files associated with it. It is possible to have business objects where the object's attributes alone are all that is required or desired. However, there will be many cases where you will want to associate external files with a business object within eMatrix. To make this association, you must check the files into the object. This is called file *checkin*.

Checking in a file allows other eMatrix users to access a file that might not otherwise be accessible. For example, assume you have a file in your personal directory. You would like to make this file accessible to your local group and the quality assurance group. In a typical computer environment, there is no way to allow selective groups to have access while denying others. You could give the file Group Access or World Access. The Group Access takes care of your immediate group but not the quality assurance group. If you give World Access to the file, ANYONE can access it, not just quality assurance. You can overcome this problem with eMatrix.

The policy definition can designate when specified persons, groups, and roles have access to an object. When an object is accessible, any files that are checked into that object are also accessible. Therefore, if a group has read access to an object, they also have read access to any files checked into the object. If the policy definition for an object includes enforced locking, no checkin for that object is allowed until the lock is released, regardless if the file being checked in is going to replace the checked-out file which initiated the lock or not.

When working with checked in files, keep in mind that the copy you check in will not change if you edit your own personal copy. While you maintain the original, any edits that you make to that file will not automatically appear in eMatrix. The only way to have those changes visible to other eMatrix users is to either check in the new version or to make the edits while you are within eMatrix (i.e., with Open for Edit).

Checking in files is controlled by the Checkin Businessobject statement.

Checkin Businessobject Statement

The Checkin Businessobject statement associates one or more files with a particular business object. To check a file into an object, you must first have Checkin privileges. If you do, you can check in the file by using the `checkin businessobject` statement.

```
checkin businessobject OBJECTID [store STORE_NAME] [format FORMAT_NAME] [unlock]
[client|server] [append] {FILENAME};
```

`OBJECTID` is the OID or Type Name Revision of the business object.

`STORE_NAME` is the name of the store where the file(s) should be checked in. This can be used to override the default store defined in the policy that determines where a governed business object's files are kept. For example, a business applications's checkin page may be used by several different companies, each with its own file store. The logic on the page

could be designed to check which company the current user is employed by, and check the file into the store assigned to that company.

FORMAT_NAME is the name of the format to be assigned to the files being checked in. (Formats are defined in the *Format Clause* of the Policy associated with the object Type.)

FILENAME is the full directory path to the files you want to associate with this business object or a directory name if you want to check in the entire directory. When including multiple filenames, separate each with a space. Filename must be specified last.

As an example of using the Checkin Businessobject statement, assume you have written procedures for assembling and disassembling a telephone. You now want to associate these procedures with the telephone object. To do that, you might write a statement similar to:

```
checkin businessobject Assembly "Phone Model 324" AD  
$MATRIXHOME/telephones/assemble324.doc  
$MATRIXHOME/telephones/disassemble324.doc;
```

In this example, it is assumed that the default format is for document files. If a default format is some other format, you would need to modify the above statement to include the Format clause. Let's assume that the two files are standard ASCII text files. Since that is not the default format, the above statement would have to be modified as:

```
checkin businessobject Assembly "Phone Model 324" AD  
format "ASCIItext"  
$MATRIXHOME/telephones/assemble324.doc  
$MATRIXHOME/telephones/disassemble324.doc;
```

If someone wants to access the files, the associated format will define the statements used to view, edit, and print the files.

Although the format identifies how the files are to be accessed, it does not necessarily mean that access is permitted. Remember that access is controlled by the policy. Also, editing access can be controlled by the use of exclusive editing locks (see *Locking and Unlocking a Business Object*).

Unlock Clause

When an object first has files checked in, it is probably not locked. However, when files are later checked out to be updated, the object should be locked to prevent other users from editing the file's contents. Checking in the file has the effect of updating the eMatrix database with the latest version of the file. But typically, the object owner will want to maintain editing control over the file. For example, the owner may be editing a CAD file and the file is still undergoing changes. If the owner checks in the file, others can monitor the process even though they cannot make changes. After the edit is complete the owner would remove the lock. The owner would include the Unlock clause in the Checkin statement. Alternatively, if access permitted, the lock could be removed with the unlock businessobject statement.

If locking is enforced in the object's policy, the checkin will fail if:

- the object is not locked
- the user performing the checkin in is not the one who locked it

When an object is locked, no files can be checked in to the object. This means that attempts to open for edit, as well as checkin, will fail. Files can be checked out of a locked object, and also opened for view.

This behavior ensures that one user is not overwriting changes made by another.

Client and Server Clauses

The **client** and **server** clauses allow programmers to specify where files are to be located when their programs are executed from a Web client (either downloaded or run on the Collaboration Server). These clauses are used to override the defaults and are ignored when executed on the desktop client.

The default file location for checkin from the Web is the Collaboration Server. Programmers can specify **client** to have the file copied from the Web client machine instead. For example:

```
mql checkin bus Assembly Wheel 0 format ascii file \tmp\text.txt;
```

would look for the file text.txt on the server, while the following would look for it on the client:

```
mql checkin bus Assembly Wheel 0 client format ascii file \tmp\text.txt;
```

The **server** clause can be specified to force the default location. For example:

```
mql checkin bus Assembly Wheel 0 server format ascii file \tmp\text.txt;
```

will yield the same results as:

```
mql checkin bus Assembly Wheel 0 format ascii file \tmp\text.txt;
```

Append clause

The append clause is used when the files should be added to the contents of the object. Without this flag, the checkin command will overwrite any files that are contained in the object, even if the file names are unique. For example:

```
checkin businessobject Assembly "Phone Model 324" AD  
format "ASCIItext"  
$MATRIXHOME/telephones/assemble324.doc;  
checkin businessobject Assembly "Phone Model 324" AD  
format "ASCIItext"  
append $MATRIXHOME/telephones/disassemble324.doc;
```

Without the append flag, the Assembly Phone Model 324 AD would contain just one file: disassemble324.doc.

Be aware that when you use the append clause to check in a file, and the business object already contains a file of the same name, the file is overwritten without a prompt for verification.

Store clause

The **store** clause can be used to override the default store (defined in the policy) that determines where a governed business object's files are kept. For example, a business application's checkin page may be used by several different companies, each with its own file store. The logic on the page could be designed to check which company the current user is employed by, and check the file into the store assigned to that company. For example:

```
checkin bus Part Sprocket 3 append store CAD c:\sprocket.cad;
```

Opening Files

Files that have been checked in to a business object using the `checkin` statement can be opened for either viewing or editing.

View—To open files and launch the appropriate application for viewing, use the following statement:

```
openview businessobject OBJECTID [format FORMAT_NAME] [file FILENAME];
```

Edit—To open files and launch the appropriate application for editing, use the following statement:

```
openedit businessobject OBJECTID [format FORMAT_NAME] [file FILENAME];
```

`OBJECTID` is the OID or Type Name Revision of the business object.

`FORMAT_NAME` is the file format in which the file has been checked in.

`FILENAME` is the name of the file.

Checking Out Files

Once a file is checked in, it can be modified by other eMatrix users who have editing access (if the object is not locked). This means that the original file you checked in could undergo dramatic changes. As the file is modified, you may want to replace your original copy with one from eMatrix, or you may want to edit the file externally. This is done by *checking out* the file.

When a business object file is checked out, a copy is made and placed in the location specified. This copy does not affect the eMatrix copy. That file is still available to other eMatrix users. However now you have your own personal copy to work on.

In some situations, a person may be denied editing access but allowed checkout privilege. This means the user may not be allowed to modify the eMatrix copy, but can obtain a personal copy for editing. This ensures that the original copy remains intact. For example, a fax template is checked in but each user can check out the template file, fill in individual information, and fax it.

Checking out files is controlled by the `Checkout Businessobject` statement, as discussed below.

Checkout Businessobject Statement

The `Checkout Businessobject` statement enables you to obtain a personal copy of the files in a particular business object. To check a file out of an object, you must first have Checkout privileges. If you do, you can check out the files by using the `Checkout Businessobject` statement. (Otherwise, eMatrix prevents you from checking out the file.)

```
checkout businessobject OBJECTID [lock] [server|client] [format FORMAT_NAME]
[file |FILENAME {,FILENAME}| all] [in VAULTNAME] [DIRECTORY];
```

`OBJECTID` is the OID or Type Name Revision of the business object. It can also include the `in VAULTNAME` clause, to narrow down the search.

`FORMAT_NAME` is the name of the format of the files to be checked out. (Formats are defined in the [Format Clause](#) of the Policy associated with the object Type.) If no format is specified, only files of the default format are checked out. If a format is specified, but no filename(s), all files of the specified format are checked out.

FILENAME is a specific file (or files) that you want to check out, or specify all to checkout all files in the specified format.

DIRECTORY is the complete directory path where you want the checked out copies. If the directory is omitted, the checked out copies are copied to the current system directory.

For example, assume you have written procedures for assembling and disassembling a telephone. After checking them in, you allow eMatrix users to edit the procedures to correct errors or ambiguities. To do this, they might write a statement similar to:

```
checkout businessobject Assembly "Phone Model 324" AD  
file assemble324.doc $MATRIXHOME\telephones;
```

After this statement is processed, a copy of the named file will appear in directory specified. If the directory is not specified, the files are copied to the current system directory.

If you have checked out an earlier version of a file to edit it, be careful not to overwrite the external file with the new checked out file. If the same file name already exists in the target directory, no error message appears in MQL. The new file will overwrite the existing file without further warning.

Lock Clause

In the Checkout statement, it is assumed that the file being checked out is unlocked to allow other users to edit the file's contents. To prevent other users from checking files in, lock it by including the keyword Lock within the Checkout statement. Only the person who locks the object will be allowed to check in files. For example, the following statement locks and checks out a file:

```
checkout businessobject Assembly "Phone Model 324" AD lock;
```

Using the lock to prevent editing is useful when you are making extensive changes to a file externally from eMatrix. While those changes are being made, you do not want to worry about any other users modifying the original file without your knowledge. When you have completed your changes, you can check the file in and remove the lock at the same time.

It is possible for a locked object to be unlocked by a user with unlock access. For example, a manager may need to unlock an object locked by an employee who is out sick.

If locking is enforced in the object's policy the object MUST be locked within the checkout command if the updated file is to be checked back in.

Server and Client Clauses

The server and client clauses allow programmers to specify where files are to be located when their programs are executed from a Web client (either downloaded or run on the Collaboration Server). These clauses are used to override the defaults and are ignored when executed on the desktop client.

By default, the checkout command executed from the Web stores files on the Web client machine. The server clause allows programmers to alternatively specify the Collaboration Server as the file location for the operation. For example, the following

statement in a Tcl program object that is run from the Web client will land the file text.txt on the client:

```
mql checkout bus Assembly Wheel 0 format ascii file text.txt \tmp;
```

while the following would look for it on the server:

```
mql checkout bus Assembly Wheel 0 server format ascii file text.txt \tmp;
```

The client clause can be specified to force the default location. For example:

```
mql checkout bus Assembly Wheel 0 client format ascii file text.txt \tmp;
```

will yield the same results as:

```
mql checkout bus Assembly Wheel 0 format ascii file text.txt \tmp;
```

Format Clause

In addition to checking out all files of the default format from the object, you can check out only specific formats of the file. Remember that files may be checked in using multiple formats. For example, a text file may be checked in using two formats that represent two different versions of a word processing program. You can check out only the file associated with the specific format by using the Format clause.

The Format clause of the Checkout Businessobject statement specifies the format of the file(s) to check out. If no format is specified, the default format is assumed. For example, the following statement checks out all files associated with the 1998 word processing program.

```
checkout businessobject "Phone Book" "Boston Region" K format 1998;
```

Moving Files

You can move files from one object to another (or another format of the same object) using the `modify bus` command. This is virtually the same thing as doing a checkout, delete file and then checkin to a new format or object. Refer to [Chapter 34, Modifying a Business Object](#), for more information.

Locking and Unlocking a Business Object

A business object can be locked to prevent other users from editing the files it contains. Even if the policy allows the other user to edit it, a lock prevents file edit access to everyone except the person who applied the lock.

Locking a business object protects the object's contents during editing. When an object is opened for editing, it is automatically locked by eMatrix, preventing other users from checking in or deleting files. However, if they have the proper access privileges, other users can view and edit attribute values and connections of the object and change its state.

A lock on an object prohibits access to the files it contains, but still allows the object to be manipulated in other ways.

But, if the checkout and edit are separate actions, the object should be manually locked. Without a lock, two people might change an object's file at the same time. With a lock, only the person who locked the object manually is allowed to check files into the object. As with an automatic lock, other users can view attributes, navigate the relationships, and even checkout an object's file. But, they cannot change the contents by checking in or deleting a file without unlocking the object.

It is possible for a locked object to be unlocked by a user with unlock access. For example, a manager may need to unlock an object locked by an employee who is out sick. See [User Access](#) in Chapter 10.

If another user has unlock privileges and decides to take advantage of them, the person who established the lock will be notified via IconMail that the lock has been removed and by which user. This should alert the lock originator to check with the *unlocker* (or the history file) before checking the file back in to be sure that another version of the same file (with the same name and format) has not been checked in, potentially losing all edits made by the unlocker.

If the object is governed by a policy which uses enforced locking, the object must be locked for files to be checked in. Users must remember to lock an object upon checkout if they intend to checkin changes, since the separate lock statement will be disabled when locking is enforced.

In the sections that follow, you will learn more about the statements that lock and unlock a business object.

Locking a Business Object

A business object can be locked using either the Lock statement or the Checkout statement. The Checkout statement is used to copy files from an object (as discussed in [Checking Out Files](#)). The Lock statement is used for general locking purposes.

If locking is enforced in the policy, the lock must be part of the checkout statement. Attempts to use the lock statement will fail.

The Lock statement places a lock on a business object:

```
lock businessobject OBJECTID;
```

OBJECTID is the OID or Type Name Revision of the business object.

When this statement is used, MQL checks to see if you have locking privileges and if there is an existing lock on the object. If you have privileges and there is no existing lock, a lock is applied to the object. (Otherwise, an error results.) After the lock is applied, only you or someone operating in your context can edit the object contents.

For example, the following statements restrict to Barbara the editing of the “Box Design” object “Bran Cereal:”

```
set context user Barbara password "Van Gogh";
lock businessobject "Box Design" "Bran Cereal" A;
```

The first statement sets the context and identifies the person placing the lock. The second statement applies the lock to the object. Unless Barbara or someone with unlocking privileges removes the lock, no one else can alter the object’s contents.

Unlocking a Business Object

A business object can be unlocked using either the Unlock statement or the Checkin statement. The Checkin statement is used when you are associating files with the object (as discussed in [Checking In Files](#)). The Unlock statement removes a lock from a business object:

```
unlock businessobject OBJECTID [comment TEXT];
```

OBJECTID is the OID or Type Name Revision of the business object.

TEXT is a special message sent to the original lock holder.

When this statement is used, MQL checks for a lock on the object. If there is an existing lock, and you are the user that locked the object, the lock will be removed from the object. If you are not the User that locked the object, MQL will check for Unlock access and will unlock the object only if you are entitled to do so.

After the lock is removed, other users may apply new locks to obtain exclusive editing access. For example, the following statements remove the exclusive editing lock from the Bran Cereal business object:

```
set context user Barbara password "Van Gogh";
unlock businessobject "Box Design" "Bran Cereal" A;
```

The first statement sets the context and identifies the person removing the lock. This must be the same person who placed the lock on the object or someone who has unlocking privileges. The second statement unlocks the object. Unless Barbara or someone with unlocking privilege removes the lock, no one else can alter the object’s contents.

Under some circumstances, it may be necessary to have someone other than the lock owner remove the lock. For example, in the case of extended illness, a manager may decide to have someone else edit the object. When the manager removes the lock, the original lock owner will receive a notice (IconMail) that the lock was removed. If desired, an additional message can be sent along with the notification. This is the purpose of the Comment clause of the Unlock statement.

Modifying the State of a Business Object

The following MQL statements control the state of a business object:

```
approve businessobject OBJECTID signature SIGN_NAME [comment VALUE];
ignore businessobject OBJECTID signature SIGN_NAME [comment VALUE];
reject businessobject OBJECTID signature SIGN_NAME [comment VALUE];
unsign businessobject OBJECTID signature SIGN_NAME|all [comment VALUE];
enable businessobject OBJECTID [state STATE_NAME];
disable businessobject OBJECTID [state STATE_NAME];
override businessobject OBJECTID [state STATE_NAME];
promote businessobject OBJECTID;
demote businessobject OBJECTID;
```

OBJECTID is the OID or Type Name Revision of the business object.

Each statement controls the movement of an object into or out of a particular state (as described in the sections that follow). A state defines a portion of an object's lifecycle. Depending on which state an object is in, a person, group, or role may or may not have access to the object. In some situations, a group should have access but is prohibited because the object has not been promoted into the next state.

Approve Businessobject Statement

The Approve Businessobject statement provides a required signature. When a state is defined within a policy, a signature can be required for the object to be approved and promoted into the next state. You provide the signature with the Approve Businessobject statement. For example, to approve an object containing an application for a bank loan, you might write this Approve Businessobject statement:

```
approve businessobject "Car Loan" "Ken Brown" A
    signature "Loan Accepted"
    comment "Approved up to a maximum amount of $20,000";
```

In addition to providing the approving signature, a comment was added to provide additional information regarding the approval in the example above. In this case, the comment informs other users that the object (Ken Brown's car loan) has been approved up to an amount of \$20,000. If the customer asks for more, the approval would no longer apply and the bank manager might reject it.

The Approve Businessobject statement provides a single approving signature. However, the Approve Businessobject signature does not necessarily mean that the object will be promoted to the next state. It only means that one of the requirements for promotion was addressed. Depending on the state definition, more than one signature may be required.

Ignore Businessobject Statement

The Ignore Businessobject statement bypasses a required signature. In this case, you are not providing an approving or rejecting signature. Instead you are specifying that this required signature can be ignored for this object.

In a policy definition, states are created to serve the majority of business objects of a particular type. This means that you may have some business objects that do not need to adhere to all of the constraints of the policy. For example, you might have a policy for developing software programs. Under this policy, you may have objects that contain programs for customer use and programs for internal use only. In the case of the internal programs, you may not want to require all of the signatures for external programs. Instead, you might be willing to ignore selected signatures since the programs are not of enough importance to warrant them.

Use the Ignore Businessobject statement to bypass a required signature. For example, assume you have a simple inventory program to track one group's supplies. The program is highly specialized for internal use and will not be used outside the company. According to the policy governing the object (which contains the program), the company president's signature is required before the program can enter the Released state for business objects outside the company. Since no one wants to bother the president for a signature, you decide to bypass the signature requirement with the following Ignore Businessobject statement. (Note that the user must have privileges to do this.)

```
ignore businessobject "Software Program" "In-house Inventory" III  
signature "Full Release"  
comment "Signature is ignored since program is for internal use only";
```

In this statement, the reason for the bypass is clearly defined so that users understand the reason for the initial bypass of the signature. As time passes, this information could easily become lost. For that reason, you should include a Comment clause in the statement even though it is optional.

The Ignore Businessobject statement involves control over a single signature. An object is promoted to the next state automatically when all requirements are met if the state was defined with the Auto Promote clause. Bypassing the Ignore Businessobject signature does not necessarily mean that the object will meet all of the requirements for promotion to the next state. It only means that one of the requirements for promotion was circumvented. Depending on the state definition, another user or condition may be required to approve the program.

Reject Businessobject Statement

The Reject Businessobject statement provides a required signature. In this case, the signature is used to prevent an object from being promoted to the next state.

For example, a bank manager may decide that more clarification is required before s/he will approve of car loan. S/he can enter this information by writing the following Reject Businessobject statement:

```
reject businessobject "Car Loan" "Ken Brown" A  
signature "Loan Accepted"  
comment "Need verification of payoff on student loan before I'll approve  
a loan up to a maximum amount of $20,000";
```

Any other users can see the reason for the rejection. Since the signature was provided, the object cannot be promoted unless someone else overrides the signature or the reason for rejection is addressed.

The Reject Businessobject statement provides a single signature. However, this signature does not necessarily mean that the object will be demoted or completely prevented from promotion to the next state. It only means that one of the requirements for promotion was denied. Depending on the state definition, another user may override the rejection.

Unsign Signature

The Unsign Signature statement is used to erase signatures in the current state of an object. Any user with access to approve, reject, or ignore the signature has access to unsign the signature.

For example the command to unsign the signature "Complete" in object Engineering Order 000234 1 is:

```
unsign businessobject "Engineering Order" 000234 1 signature Complete;
```

Errors will occur under the following conditions:

- Attempts to unsign a signature not yet signed.
- Attempts to unsign all signatures if all are not signed, any that are signed; however, will be unsigned.
- Attempts to unsign a non-existent signature.
- Attempts to unsign a signature without access.

Disable Businessobject Statement

The Disable Businessobject statement holds an object in a particular state indefinitely. When a business object is in a disabled state, it cannot be promoted or demoted from that state. Even if all of the requirements for promotion or demotion are met, the object cannot change its state until the state is enabled again.

Use the Disable Businessobject statement to disable a business object within a state:

```
disable businessobject OBJECTID [state STATE_NAME];
```

OBJECTID is the OID or Type Name Revision of the business object.

STATE_NAME is the name of the state in which you want to freeze the object. If you want to disable the current state, the State clause is not required.

For example, to disable an object containing a component design for an assembly, you could write a Disable Businessobject statement as:

```
disable businessobject "Component Design" "Bicycle Seat R21" A  
state "Initial Release";
```

Labeling an object as disabled within a state does not affect the current access. This means that the designer can continue to work on the object in that state.

Enable Businessobject Statement

The Enable Businessobject statement reinstates movement of an object. When a business object is in a disabled state, it cannot be promoted or demoted from that state. If you then decide to allow an object to be promoted or demoted, you must re-enable it using the Enable Businessobject statement.

When an object is first created, all states are enabled for that object. If a manager or other user with the required authority decides that some states should be disabled, s/he can

prevent promotion with the Disable Businessobject statement. After the statement is processed, an object will remain trapped within that state until it is enabled again.

Use the Enable Businessobject statement to enable a business object within a state:

```
enable businessobject OBJECTID [state STATE_NAME];
```

OBJECTID is the OID or Type Name Revision of the business object.

STATE_NAME is the name of the state in which you want to enable the object. If you want to enable the current state, the State clause is not required.

For example, assume Component Design is disabled. All of the conditions for an Initial Release state have been met and the manager decides it is ready for promotion into the Testing state. Before the object can be promoted, however, it must be enabled. The manager can do this with the following statement:

```
enable businessobject "Component Design" "Bicycle Seat R21" A  
state "Initial Release";
```

The object is enabled and available for promotion or demotion.

Enabling an object does not have an effect on the remaining states. For example, the Testing state could be disabled at the same time the Initial Release state was disabled. When the Initial Release state is enabled, the object can be promoted into the Testing state. However, once it is in this new state, it again can no longer be promoted or demoted. It will remain in the Testing state until it is enabled for that state.

Override Businessobject Statement

The Override Businessobject statement turns off the signature requirements and lets you promote the object. Since a policy is a generic outline that addresses the majority of needs, it is possible to have special circumstances in which a user, such as a manager, may decide to skip a state rather than have the object enter the state and try to meet the requirements of the state. This is done using the Override Businessobject statement.

Use The following syntax to write an Override Businessobject statement:

```
override businessobject OBJECTID [state STATE_NAME];
```

OBJECTID is the OID or Type Name Revision of the business object.

STATE_NAME is the name of the state that you want to skip. If you want to override the current state, the State clause is not required.

For example, assume you have a component that has undergone cosmetic changes only. This component is needed in Final Release although the policy dictates that the component must go through Testing before it can enter that state. Since the changes were cosmetic only, the manager may decide to override the Testing state so that the users can access the component sooner. This could be done by entering a statement similar to:

```
override businessobject "Component Design" "Bicycle Seat R21"  
state Testing;
```

Now, assume that the object is currently in the Initial Release state and a promotion would place it in the Testing state. How does the Override Businessobject statement affect the object when it is promoted? The object will be promoted directly from the Initial Release state into the Final Release state.

Promote Businessobject Statement

The Promote Businessobject statement moves an object from its current state into the next state. If the policy specifies only one state, an object cannot be promoted. However, if a policy has several states, promotion is the means of transferring an object from one state into the next.

The order in which states are defined is the order in which an object will move when it is promoted. If all of the requirements for a particular state are met, the object can change its state with the Promote Businessobject statement.

```
promote businessobject OBJECTID;
```

OBJECTID is the OID or Type Name Revision of the business object.

For example, the following statement would promote an object containing an application for a bank loan:

```
promote businessobject "Car Loan" "Ken Brown" A;
```

An object cannot be promoted if:

- Its current state is disabled.
- The signature requirements were not met or overridden (ignored).
- There are no other states beyond the current state.

Demote Businessobject Statement

The Demote Businessobject statement moves an object from its current state back into its previous state. If the policy has only one state or is in the first state, an object cannot be demoted. However, if a policy has several states, demotion is the means of transferring an object backward from one state into the previous state.

If an object reaches a state and you determine that it is not ready for that state, you would want to send the object back for more work. This is the function of the Demote Businessobject statement:

```
demote businessobject OBJECTID;
```

OBJECTID is the OID or Type Name Revision of the business object.

For example, assume you have a component that has undergone cosmetic changes only. The manager decides to send it into Final Release without sending it through testing. Now the manager finds out that the new paint trim might weaken the plastic used in the seat. Therefore, the manager decides to demote the object back into the Testing state. This could be done by entering a statement similar to:

```
demote businessobject "Component Design" "Bicycle Seat R21" A;
```

When using the Demote Businessobject statement, an object cannot be demoted if:

- The current state is disabled.
- The signature requirements for rejection were not met or overridden (ignored).
- There are no other states prior to the current state.

36

Working With History

History Overview	752
Adding a Custom History Record for Business Objects	753
Selecting History Entries	754
Excluding History when Printing or Expanding	754
Selecting History.....	754
Deleting History	758
delete history clause	758
Important Notes	762
Enable/Disable History.....	763

History Overview

eMatrix provides a history for each business object, detailing every activity that has taken place since the object was created.

An object's historical information includes:

- The type of activity performed on the object, such as create, modify, connect, and so on.
- The user who initiated the activity.
- The date and time of the activity.
- The object's state when the activity took place.
- Any attributes applicable when the activity took place.

You can add a customized history through MQL to track certain events, either manually or programmatically.

History records can be selected when printing or expanding business object, set, or connection information using the MQL `select` clause. In addition, when printing all information about a business object, set, or connection, history records can be excluded.

System administrators only can purge the history records of a business object or connection via MQL. In addition, *all* history records of a business object or connection can be deleted with one command.

History can be turned off in a single session by a System Administrator for the duration of the session or until turned back on. In addition, if an implementation does not require history recording, or requires only custom history entries, eMatrix "standard" history can be disabled for the entire system. Turning history recording off can improve performance in very large databases, though certain standards may require that it is turned on.

Adding a Custom History Record for Business Objects

You can add a customized history record through MQL to track certain events, either manually or programmatically. Two parts of the entry are definable: the Tag, and the Comment. The Tag appears at the beginning before the hyphen. Then the user/time/date/current stamp is automatically made, followed by the comment defined by the implementer. The MQL syntax for defining a history entry is:

```
modify bus OBJECTID add history VALUE [comment VALUE];
modify connection ID add history VALUE [comment VALUE];
modify connection bus OBJECTID from|to OBJECTID relationship NAME add history VALUE
[comment VALUE];
```

For example, the following command could be added to an MQL/Tcl program which backs up the database:

```
modify bus $OBJECT add history Backup comment Backup was completed to tape #12345;
```

The history entry for the above would look something like:

```
(Backup) - user: billy time: Mon Mar 30, 1998 11:28:15 AM Eastern Standard Time state:
planned comment: Backup was completed to tape #12345.
```

All custom history event entries are enclosed in parentheses in order to distinguish them as such.

Custom history entries do not respond to the history off or set system history off commands.

Selecting History Entries

History records of either business objects or connections can be selected with the `select` clause of the following MQL commands:

<code>print businessobject</code>	<code>expand businessobject</code>
<code>print set</code>	<code>expand set</code>
<code>print connection</code>	

In addition, when using the above commands to print all information about a business object, set or connection, history records can be excluded.

Excluding History when Printing or Expanding

When all information about a business object or connection is printed with the `print` commands listed above, everything about the object(s) or connection, including its history, is listed.

The `!history` clause allows you to exclude the history of a business object or connection, which can be quite lengthy, when printing all the other information. For example, the following command:

```
print bus Assembly RB45621 A !history;
```

prints out all information about the business object *except* its history.

Selecting History

History can be selected from either business objects or connections, in a manner similar to selecting the owner or current state of an object. For more information on the MQL select mechanism refer to the *Select Expressions* appendix in the *eMatrix Navigator Guide*.

To select history from a business object use the following syntax:

```
print bus OBJECT select history.ITEM [history.ITEM];
```

Where:

OBJECT is the Type, Name, and Revision of the business object or its object ID.

ITEM can be one of the following:

EVENT
time
user
state

To select history from a connection, use one of the following:

```
print connection ID select history.ITEM;  
Or  
expand bus OBJECT select relationship history.ITEM;
```

Where ITEM can be:

EVENT
time
user

ID is the identification of the connection as returned with the select connection ID print command.

The second command listed will actually return the history of all connections on the specified OBJECT.

Each ITEM clause is described in the sections that follow.

history.EVENT clause

The `history.EVENT` clause can also be used on a business object or connection. It returns a list of history records of the EVENT event type. For business objects, EVENT can be one of the following:

approve	custom	moveto
changename	delegate	override
changeowner	demote	promote
changepolicy	disable	purge
changetype	enable	reject
changevault	ignore	removedoid
checkin	lock	removefile
checkout	modify	revise
connect	modifyattribute	schedule
create	movedoid	undelegate
	movefrom	unlock

For a connection, EVENT can be:

changetype	freezethaw	modifyattribute
create	modify	purge
custom

Example 1

For example, if a user enters:

```
print bus Assembly PR6792 A select history.modify;
```

the following is output:

```
modify - user: patrick time: Mon Aug 6, 1998 5:50:11 PM  
state: Assigned description: test  
modify - user: sam time: Mon Aug 6, 1998 6:50:11 PM state:  
Assigned description: test1
```

```
modify - user: diane time: Mon Aug 6, 1998 7:50:11 PM state:  
Assigned description: test2  
modify - user: ted time: Mon Aug 6, 1998 8:50:11 PM state:  
Assigned description: test3
```

Example 2

You may want to find the date when a signature has been satisfied. Use a statement similar to the following:

```
print bus ECR 000122 "" select history.approve;
```

This statement returns:

```
business object ECR 000122 history.approve = approve - user: Cole time:  
Sun Sep 5, 1999 1:24:50 PM CDT state: Approvals signature: Scrap Approver 1  
comment:
```

If you have multiple signatures on specific states, you will have to parse the data for a particular date and signature.

Example 3

To return a list of customized history entries created with the `add history VALUE [comment VALUE]` clause of the `modify bus` statement, use the `custom` event. For example:

```
print bus 2340988 select history.custom;
```

might return:

```
(Backup) - user: billy time: Mon Mar 30, 1998 11:28:15 AM  
Eastern Standard Time state: planned comment: Backup was  
completed to tape #12345.
```

Note that `freezethaw` is one event that can be selected and will return both freeze and thaw entries.

history.time clause

The `history.time` clause can be used on a business object or connection to return a list of the timestamps of every history record. For example:

```
print connection 1234568 select history.time;
```

returns a list of the different times that the connection was updated in the database:

```
time: Fri, Jan 2, 1998 1:48:41 PM  
time: Thurs, Feb 6, 1998 2:20:13 PM  
time: Wed, April 8 1998 10:15:12 AM
```

history.user clause

The `history.user` clause can be used on a business object or connection. It essentially gives a list of all users who have operated on the object or connection. For example, the following:

```
print bus Manual NewBook 1 select history.user;
```

may output:

```
user: sue  
user: tim  
user: jerry
```

history.state clause

The `history.state` clause can be used only on a business object or set. It returns a list of all states the object was in when operations were performed on it. For example:

```
print bus 2340988 select history.state;
```

gives a list of all states the business object was in after every change to the business object:

```
state: Proposed  
state: Assigned  
state: Described
```

Deleting History

System administrators only can purge the history records of a business object or connection via MQL. History records can be deleted based on:

- the type of event (for example, checkout, checkin);
- the user who performed the event (for example, angie);
- the date the operation took place (for example, on, before, or after a specified date).

In addition, *all* history records of a business object or connection can be deleted with one command. Users can optionally write the purged entries to a file. The purge history event itself is recorded in history.

While the various forms of the command provide flexibility and control over exactly which history records are purged, very complicated variations are not supported. This is to ensure that accidental deletions of important historical events do not occur. In other words, as the criteria for deletion becomes more complex, more delete history statements will be required.

In the History of an object, other objects are sometimes referred to. This is an issue if “Show” access has been denied for a particular user or object in the database. The performance impact of determining whether the current user has access to see the Type, Name and Revision of such objects would be significant and unavoidable. Individual history records can be deleted using the `delete history` clause of the `modify businessobject` or `modify connection` command. This can be used in action triggers to remove such records.

delete history clause

The delete history clause of the `modify businessobject` or `modify connection` clause can be used alone, or with other refining ITEM clauses. If there are no ITEMS specified, then *all* history records associated with the business object or connection are deleted. The syntax is:

```
modify businessobject OBJECT delete history [ITEM {ITEMS}];
```

OR

```
modify connection ID delete history [ITEM {ITEMS}];
```

Where:

OBJECT is the Type, Name, and Revision of the business object or its object ID.

ID is the identification of the connection as returned with the `select connection ID print` command.

ITEM can be from the list below:

event EVENT		
by USER		
on DATE	before	after

keep	last
output FILENAME	

Notice that the keywords "keep" and "last" are mutually exclusive. Also, only one form of the DATE item is allowed in a single statement.

For example:

```
modify bus Document P0567932 1 delete history;
```

deletes all history records and adds a history entry similar to the following:

```
History Records Purged by ted on 11/9/98 12:02:03 PM.!!!
```

Each ITEM that can be used with the history delete clause is described in the sections that follow.

event EVENT item

All history entries that log a specific event can be deleted from a business object or connection using the event EVENT item. EVENT is a database event that is logged in history. For business objects, EVENT can be from the following list::

approve	custom	moveto
changename	delegate	override
changeowner	demote	promote
changepolicy	disable	purge
changetype	enable	reject
changevault	ignore	removedoid
checkin	lock	removefile
checkout	modify	revise
connect	modifyattribute	schedule
create	movedoid	undelegate
	movefrom	unlock

EVENT for connections can be from the following list:

changetype	custom	modify
create	freezethaw	purge

Notice that the purge history event record itself can be deleted; however, doing so will be generate a new purge history record.

A list of events can be specified, separated by a space and a comma. For example:

```
modify bus Document P0567932 1 delete history event changetype, changename;
```

deletes all entries of these types. And:

```
modify connection 35894008 delete history event freezethaw;
```

removes all freeze and thaw history entries on the connection. Note that freezethaw is one selectable EVENT.

To purge customized history entries created with the add history VALUE [comment VALUE] clause of the modify bus statement, use the custom event. For example:

```
modify bus 2340988 delete history custom;
```

by USER item

All history entries that log events performed by a particular user can be deleted from a business object or connection using the by USER item.

USER is the person listed in the history entry as the user who performed the operation, and so is, or was, a valid eMatrix Person.

A list of persons can be specified, separated with a space and a comma. Also, the by USER clause can be used with or without the keyword by and in conjunction with other ITEMS. A few examples:

```
modify bus Document PO567932 1 delete history by chris, tom;
```

deletes the records of all events performed by either chris or tom from the object's history.

```
modify bus Document PO567932 1 delete history event changetype by chris, tom;
```

deletes all changetype history events performed by either christie or tom.

DATE items

History entries can be deleted from a business object or connection based on when the event occurred using one of the following DATE items.

on DATE
before DATE
after DATE

DATE is the timestamp in the history entry, and may include the time of day, or just the date of the event. If the time of day is not included, then the input is considered a date. Otherwise the timestamp is taken into account. For example:

```
modify bus Document PO567932 1 delete history date 11/04/98;
```

deletes the record of all events that occurred on November 4, 1998. On the other hand:

```
modify bus Document PO567932 1 delete history before "11/04/98 12:00:00 PM EST";
```

deletes all entries for events that occurred before November 4, 1998 at 12:00:00 PM EST. The after keyword can be used in the same manner.

Quotes must be used when including the time of day in the DATE.

Dates and times can be entered in the command as specified in the initialization file. Refer to the *eMatrix Installation Guide* for more information on date/time formats.

Date items can be used with or without the other ITEMS listed, but only one date item is allowed in a single command. For example:

```
mod bus Document PO567932 1 delete history event checkin by sue after 3/17/01;
```

Only one date ITEM is allowed in a single command.

keep item

The keep item is used in conjunction with the other ITEMS and provides a way of reversing what is specified. The ITEMS that follow keep, are NOT deleted, but all other records are. The one exception is that any purge history records are always kept.

In addition to what is specified with the keep clause of the delete history statement, purge history entries are always kept, unless ALL history is deleted.

Keep can be used with any of the other ITEMS except last, but it must be specified immediately following the delete history clause. For example:

```
modify bus Document PO567932 1 delete history keep event create;
```

deletes all records except the creation (originated) entry.

```
modify bus Document PO34675 1 delete history keep event checkin by bill on November 5, 1998;
```

deletes all records except those for checkin events performed by bill on the date specified.

When used, keep must be specified immediately following the delete history clause, and may NOT be used in conjunction with last.

last item

The last item is used in conjunction with the other ITEMS (except for keep) to purge the most recent history records that meet the criteria. When used, last must be specified immediately following the delete history clause. For example:

```
modify bus Document PO34675 1 delete history last event checkin by bill;
```

deletes only one entry - for the last checkin event performed by bill.

When used, last must be specified immediately following the delete history clause, and can NOT be used in conjunction with keep.

output FILENAME item

The purged history entries can be written to a text file using the output FILENAME item. In this manner, history can be archived.

FILENAME is the name of a file to create with the purged history records. Optionally, a directory path can be included.

For example:

```
modify bus Document PO34675 delete history event checkin output history.txt;
```

deletes all “checkin” records and stores them in the file “history.txt.” The file will be saved in MATRIXHOME unless a path is specified. For example, on a PC, the following could be used:

```
modify bus Document PO34675 delete history event checkin output  
"d:\archive\history.txt";
```

On UNIX it might be:

```
modify bus Document PO34675 delete history event checkin output "/home/archive/  
history.txt";
```

Important Notes

- Every time a purge is performed for either business objects or connections, a history record is added which says:

```
history = purge - user: USER time: TIMESTAMP 'History Records Purged'
```

These purge entries are deleted only when all history is deleted, or when explicitly deleting them, with something like the following:

```
modify bus Document PO34675 1 delete history event purge;
```

However, if you use the keep clause to save all “modify” records, for example, all “modify” and “purge” records will actually be kept.

- If NO ITEMS are specified, then ALL history records associated with a business object or connection are purged, including any purge history entries.
- Once history records are purged, they are completely deleted from the database. So a system administrator should take extra care before purging records. If the output clause is used, the text file could be checked into an object, but entries cannot be imported back into the history log.
- History deletion operations cannot be strung together into one single command. For example, THE FOLLOWING IS NOT SUPPORTED:

```
modify bus TYPE NAME REVISION delete history event lock keep event lock by Jo;
```

The delete history clause has been kept as simple as possible, to ensure that unwanted deletions do not occur. To actually do something like the above, (delete only lock records but keep lock records performed by one user), a different selection criteria must be formulated and accomplished using more than one command.

Enable/Disable History

History can be disabled for a session, or until re-enabled. This improves performance when creating or importing many objects. This is also useful when bulk loading business objects, or if a different history logging mechanism is implemented.

System Administrators can execute the following MQL command.

```
history off;
```

After this command is executed, events that occur on the local machine do not cause a history record to be logged in the database. This command affects only the local machine; concurrent user's sessions are not affected. In addition, subsequent sessions on the local machine will have history enabled by default.

To enable history recording for the session again, a System Administrator should use:

```
history on;
```

MQL also allows history to be enabled or disabled with the use of a toggle command. Depending on the current setting, history can be enabled/disabled using the same command.

To enable/disable history as a toggle, a System Administrator should use:

```
history;
```

When the recording of history is turned off in a program object, it is important that it gets turned back on. While this may be done at the bottom of the program object's code section, history recording is re-enabled automatically when a top-level program ends its execution even if the code is exited before reaching the command that turns it back on (either successfully or in error).

When used with the ADK MQLCommand class, an MQL session lasts for only the duration of one command and not throughout the ADK program's session. This means that the `history off` command has no effect, since the MQL session ends and so history is turned back on. This is the design intent, to ensure that history is not inadvertently turned off for longer than intended.

Since logging history affects performance as well as the size of the database, in some implementations it may be desirable to turn history off permanently. Refer to [Controlling System-wide Settings](#) in Chapter 3 for more information.

The system history setting should not be issued within program objects, since it affects all users. In these cases, the temporary MQL history off command should be used instead.

Working With Sets

Set Defined	766
Understanding Differences	766
Creating a Set	767
Copying or Modifying a Set Definition.....	770
Copying (Cloning) a Set Definition.....	770
Modifying a Set Definition.....	770
Expanding Objects Within Sets.....	772
Relationship Expressions	772
Expanding From	772
Expanding To.....	773
Relationship Clause.....	774
Type Clause.....	774
Recurse Clause	775
Set Clause	775
Dump and Output Clauses	776
Tcl Clause.....	776
Terse Clause	776
Viewing Set Definitions	777
Select Clause.....	777
Dump Clause	778
Recordseparator Clause.....	778
Tcl Clause.....	778
Output Clause.....	779
Expressions on Sets	780
Count	780
Sum	780
Maximum, Minimum, Average	781
Median	781
Standard Deviation	781
Correlation	782
Deleting a Set.....	783

Set Defined

A *set* is a logical grouping of business objects created by an individual user. Sets are often the result of a query made by a user. For example, the user may want to view all drawings created for a particular project or find information about objects having a particular attribute or range of attributes. While sets can be manually constructed based upon the needs and desires of the individual, queries are a fast means of finding related business objects.

The contents of a set can be viewed at any time and objects can be added or deleted from a set easily. However, a user has access only to sets created while in session under his/her context.

Understanding Differences

It is important to realize that sets are not the same as connections between business objects. Connections also group business objects together in a window for users to analyze; however, connections are globally known links rather than local links.

Business Object Connection	Set
Created only if the policy permits.	Created without any special privileges.
Available to view by all users who have access to the objects.	Available to view at any time by the person who created the set.
Valuable to all users who have access to the objects.	Valuable only within the context of the person who created it.

Creating a Set

Sets are often the result of a query. To save the contents of a query in a set, see [Evaluating Queries](#) in Chapter 38.

To manually define a set from within MQL, use the Add Set statement:

```
add set NAME [ADD_ITEM {ADD_ITEM}] ;
```

NAME is the name of the set you are defining.

All sets must have a unique name assigned within a given context. If you duplicate a name, an error message is displayed. Although a single user cannot duplicate a set name, different users can use the same name. Remember that sets are local to the context of individual users. This means that several users could each have a set called “Current Project.” However, as you change context from one user to another, the contents of “Current Project” will most likely vary.

The set name is limited to 127 characters.

ADD_ITEM is an Add Set clause that provides more information about the set you are defining. None of the clauses is required. The Add Set clauses are:

member businessobject OBJECTID [in VAULT]
member businessobject ID
BUS_OBJ_COLLECTION_SPEC
[! not]hidden
property NAME [to ADMINTYPE NAME] [value STRING]

OBJECTID is the OID or Type Name Revision of the business object.

Each clause and the arguments they use are discussed in the sections that follow.

Member Clause

After assigning a set name, specify the business objects to include in the set. Business objects can be referenced by their complete business object name/specification or by their ID:

```
member businessobject TYPE NAME REVISION [in VAULT]
```

Or

```
member businessobject ID
```

When using the complete specification, you *must* include the object type, the name of the object, and the revision designator in this order. Note that some business objects may not have a revision designator. If a revision designator was not assigned to an object, “” (double quotes) are required in MQL.

The following are examples of complete business object names:

businessobject Assembly "Keyboard" ""
businessobject Assembly "Monitor" E
businessobject Assembly Computer AA

To group these business objects into a set, you can write a statement similar to the following:

add set "Computer Set" member businessobject Assembly "Keyboard" "" member businessobject Assembly "Monitor" E member businessobject Assembly Computer;
--

When this statement is processed, a set called “Computer Set” is created. It contains three business objects. These objects will appear grouped in a window whenever the set name is referenced.

You can also optionally specify the vault in which the business object is held. When the vault is specified, only the named vault needs to be checked to locate the business object. This option can improve performance for very large databases.

When business objects are created they are given an internal ID. As an alternative to TYPE NAME REVISION, you can use this ID when indicating the business object to be acted upon. The ID of an object can be obtained by the use of the print businessobject... selectable statement. Refer to [Viewing Business Object Definitions](#) in Chapter 34 for information.

Business Object Collection

A BUS_OBJ_COLLECTION_SPEC can be added to a set. This allows you to add sets, queries, temporary sets, temporary queries and expansions of business objects to sets. Any of these items can be used in combinations covering multiple levels of complexity using the binary operators and, or and less. For complete details, see Business Object Collection in chapter 34, *Working With Queries*.

Hidden Clause

You can specify that the new set object is “hidden” so that it does not appear in the Set chooser in eMatrix. Users who are aware of the hidden set’s existence can enter its name manually where appropriate. Hidden objects are accessible through MQL.

Property Clause

Integrators can assign ad hoc attributes, called Properties, to the set. Properties allow associations to exist between administrative definitions that aren’t already associated. The property information can include a name, an arbitrary string value, and a reference to another administration object. The property name is always required. The value string and object reference are both optional. The property name can be reused for different object references, that is, the name joined with the object reference must be unique for any object that has properties.

add set NAME property NAME [to ADMINTYPE NAME] [value STRING];
--

For additional information on properties, see *Overview of Administration Properties* in Chapter 24.

Copying or Modifying a Set Definition

Copying (Cloning) a Set Definition

After a set is defined, you can clone the definition with the Copy Set statement. This statement lets you duplicate person definitions with the option to change the value of clause arguments:

```
copy set SRC_NAME DST_NAME [COPY_ITEM {COPY_ITEM}] [MOD_ITEM {MOD_ITEM}];
```

SRC_NAME is the name of the person definition (source) to be copied.

DST_NAME is the name of the new definition (destination).

COPY_ITEM can be:

fromuser USERNAME	USERNAME is the name of a person, group, role or association.
touser USERNAME	
overwrite	Replaces any sets of the same name belonging to the user specified in the touser clause.

MOD_ITEMS are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

Modifying a Set Definition

After a set is defined, you can change its content with the Modify Set statement. This statement lets you add or remove business objects from the set:

```
modify set NAME [MOD_ITEM {MOD_ITEM}];
```

NAME is the name of the set to modify.

MOD_ITEM is the type of modification to make. Each is specified in a Modify Set clause, as listed in the following table. Note that you need to specify only fields to be modified.

Modify Set Clause	Specifies that...
add businessobject TYPE NAME REVISION [in VAULT]	The named business object is added to the set.
add businessobject ID	The business object with the specified ID is added to the set.
remove businessobject TYPE NAME REVISION [in VAULT]	The named business object is removed from the set.
remove businessobject ID	The business object with the specified ID is removed from the set.
hidden	The hidden option is changed to specify that the object is hidden.
nothidden	The hidden option is changed to specify that the object is not hidden.
property NAME [to ADMINTYPE NAME] [value STRING]	The named property is modified.

Modify Set Clause	Specifies that...
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.

As you can see, each modification clause is related to the clauses and arguments that define the person.

You can use the Modify Set statement to add another business object to the set or remove one. To remove a set entirely, use the Delete Set statement, as described in [Deleting a Set](#). To change the name of the set, remove the current set and create a new set.

For example, assume you have a set named “Product Comparison.” You want to add two new business objects and remove an old one. You might write a statement similar to the following:

```
modify set "Product Comparison"
  add businessobject "Scent Formula" "Perfume P39" B;
  add businessobject "Scent Formula" "Perfume P40" A;
  remove businessobject "Scent Formula" "Scent P13" D;
```

If you are unable to open a set created in an earlier version of eMatrix, see your Business Administrator for assistance.

Expanding Objects Within Sets

[Viewing Business Object Definitions](#) in Chapter 34 describes how you can use the Expand Businessobject statement to display connections made between two objects. The Expand Businessobject statement enables you to see other objects connected to the one you have. Depending on the form of this statement you use, you can display the objects that lead to it, lead from it, meet selected criteria, or are of a specific type.

When using the Expand Businessobject statement, you are working with a single business object. But what if you wanted to expand multiple objects? In that case, you might use the Expand statement, as described in this section. The Expand statement also lets you search for connections that meet the search criteria. However, rather than expanding only one business object, you can expand each of the objects contained within a set.

```
expand set NAME [from|to] [relationship PATTERN [type PATTERN]]  
[RECURSE [to N|all]] [into|onto set NAME][dump] [tcl] [output FILENAME][terse];
```

NAME is the full specification of a set name.

PATTERN is the pattern of the relationship or type name.

FILENAME is the name of an output file.

Like the Expand Businessobject statement, the Expand Set statement has four forms: From, To, Relationship, and Type. Since MQL will use the names of the objects from the set, you do not need to specify an object with this statement. MQL will expand each set depending on the keyword (and values) that you specify.

In addition to a list of business objects that meet the specified criteria, the Expand Businessobject statement provides information about the connections. The level number of the expansion and the relationship type of the connection are provided along with the business object name, type, and revision.

In addition, you can use the Recurse argument to indicate that you want to expand the business objects connected/related to the initially specified business object.

Refer to [Evaluating Queries](#) in Chapter 38, for a description of the Evaluate statement to *find on a set*.

Relationship Expressions

Select expressions are applied directly to relationship instances, enabling selection of its relationship attribute values for use in Indented Tables and Visual Cues.

For example, the following can be used in a table column definition:

```
relationship [Drawing].attribute[Quantity] =
```

Expanding From

When you use the From form of the Expand Set statement, you expand away from the starting object. This gives you all the related objects that are defined as the TO ends in a relationship. Consider the starting object as the tail of the arrow and you are looking for all the arrow heads. The From form uses the syntax:

```
expand set NAME from [relationship PATTERN] [type PATTERN] [RECURSE_CLAUSE]  
[SET_CLAUSE|DUMP_CLAUSE];
```

The From form of the Expand clause returns all objects connected from the starting object (the arrow points out) regardless of the relationship used.

NAME is the full specification of a set name.

PATTERN is the pattern of the relationship or type name.

RECURSE_CLAUSE expands the business object from which the initially specified business object is connected (as described in [Recurse Clause](#)).

SET_CLAUSE places the output of the expand statement into a set. All forms of the Expand Set statement use a Set clause (as discussed in [Set Clause](#)).

DUMP_CLAUSE specifies a general output format for the expanded information (as discussed in [Dump and Output Clauses](#)).

For example, assume you have a set of components which are used in larger assemblies. You need a list of all objects that each part goes into. The command would be:

```
expand set Components from;
```

This might produce a list of objects such as a calculator, telephone, VCR, and so on. Since no other criteria is specified, the From form of the Expand Set statement gives you ALL objects that occupy the TO end of a relationship definition.

Expanding To

When the To form of the Expand Set statement is used, the set's business objects are again used as the starting point. However, now it is assumed that each object is defined as the TO end of the relationship definition. Therefore, you are looking for all objects that lead *to* the named object. These are the objects defined as the FROM ends in a relationship definition.

The To form of the Expand Set statement uses the syntax:

```
expand set NAME to [relationship PATTERN] [type PATTERN] [RECURSE_CLAUSE]  
[SET_CLAUSE|DUMP CLAUSE];
```

The To form of the Expand clause returns all objects connected to the starting object (the arrow points in) regardless of the relationship used.

NAME is the full specification of a set name.

PATTERN is the pattern of the relationship or type name.

RECURSE_CLAUSE expands the business objects to which the initially specified business object is connected (as described in [Recurse Clause](#)).

SET_CLAUSE places the output of the expand statement into a set. All forms of the Expand Businessobject statement use a Set clause (as described in [Set Clause](#)).

DUMP_CLAUSE specifies a general output format for the expanded information (as discussed in [Dump and Output Clauses](#)).

Using the To form is useful when you want to work backwards. For example, you may want to know what components make up each assembly in a defined set. For example:

```
expand set "Assembly Components" to;
```

This might give you objects that contain buttons, plastic housings, printed circuit boards, etc. All related objects defined as the FROM connection end are listed.

Relationship Clause

The Relationship form of the Expand Set statement displays all objects connected in a specific relationship. This is useful when you are working with an object that may use multiple relationship types. If the starting object can only connect with one relationship, this form has the effect of listing all the connection ends used by the starting object. These ends may be defined as a TO end or a FROM end—it does not matter. Only the relationship type is of importance.

The Relationship clause uses the syntax:

```
expand set NAME [from|to] [relationship PATTERN] [type PATTERN] [RECURSE_CLAUSE]  
[SET_CLAUSE|DUMP CLAUSE];
```

The Relationship clause of the Expand Set statement returns all objects connected to the starting object with a specific relationship. The command can be made more particular by specifying the direction of the relationship and/or the types of objects to return.

NAME is the full specification of a set name.

PATTERN is the pattern of the relationship or type name. The pattern may consist of a single relationship or type name or it may include wildcards.

RECURSE_CLAUSE expands the business objects related to the initially specified business object (as described in [Recurse Clause](#)).

SET_CLAUSE places the output of the expand statement into a set. All forms of the Expand Businessobject statement use a Set object (as described in [Set Clause](#)).

DUMP_CLAUSE specifies a general output format for the expanded information (as described in page [Dump and Output Clauses](#)).

For example, assume you have a set of drawing objects. These drawings may use a number of relationships such as a User Manual Relationship, Design Relationship, Marketing Relationship, Drawing Storage Relationship, etc. You may want to examine all objects that use a particular type of relationship. For example, you might want a list of all objects that have used each drawing in some marketing way. To do this, you might enter a statement similar to:

```
expand set "Drawing Collection" relationship "Marketing Relationship";
```

This statement expands each object contained within the set named “Drawing Collection.” As each object is expanded, MQL lists any related objects that have the Marketing Relationship type. It searches through all the object connections for the Marketing Relationship type. If MQL finds a relationship with the type that uses the object being expanded, the other connection end is displayed and the end’s directional relationship is identified. It does not matter whether the related objects are defined as the FROM end or the TO end of the relationship. Only the relationship type is of importance.

Type Clause

The Type clause of the Expand Set statement displays all related objects of a specific object type. This is useful when you are working with a set of objects that may be connected to multiple object types. If the starting object can only connect with one object type, this form is similar to the Relationship form using a wildcard pattern. The Type form uses the syntax:

```
expand set NAME [from|to] [relationship PATTERN] [type PATTERN] [RECURSE_CLAUSE]  
[SET_CLAUSE|DUMP CLAUSE];
```

NAME is the full specification of a set name.

PATTERN is the pattern of the relationship or type name. The type pattern may consist of an object type name or it may include wildcards for locating type names.

RECURSE_CLAUSE expands the business objects related to the initially specified business object type (as described [Recurse Clause](#)).

SET_CLAUSE places the output of the expand statement into a set. All forms of the Expand Businessobject statement use a Set clause (as discussed in [Set Clause](#)).

DUMP_CLAUSE specifies a general output format for the expanded information (as described in [Dump and Output Clauses](#)).

For example, assume you have a set that contains training course objects. To each object, you might have related objects that are of type Evaluation, Student, Materials, Locations, etc. You may want to examine all Evaluation type objects in order to trace a course's progress in meeting student needs. To do so, enter a statement similar to:

```
expand set "Training Courses" type Evaluation;
```

This statement expands each of the objects contained in the "Training Courses" set. It lists any related objects that have the Evaluation type. Those objects might belong to multiple relationship types (such as "Professional Evaluation Relationship" or "Student Evaluation Relationship"). It does not matter if the related objects are defined as the FROM end or the TO end of the relationship. Only the object type matters in the location and display of the output objects.

Recurse Clause

Once you have a list of related objects, you may also want to expand these objects. The Recurse clause of the Expand Businessobject statement expands through multiple levels of hierarchy by applying the Expand statement to each business object found:

```
recurse to [N|all]
```

N is any number indicating the number of levels that you want to expand.

all indicates that you want to expand all levels until all related business objects are found.

Set Clause

Once you have a list of related objects, what do you do with them? In some cases, you can simply search for a particular object and you will not need to reference the output object again. In that case, you might want to display the expansion output on your terminal. However, in other cases, you may want to capture the output and save it. That is the reason for the Set clause of the Expand Set statement.

The Set clause uses the following syntax:

```
into|onto set SET_NAME
```

SET_NAME is a valid name value that will be assigned to the set.

When the Set clause is included within the Expand Set statement, the related objects are placed within a set and assigned a set name. This set may already contain values or it may be a new set created for the purpose of storing this output.

When it is an existing set, the previous values are either replaced or added onto depending on the keyword you use to begin the Set clause:

into	The existing set contents are discarded and only the current output is saved.
onto	The new output is appended onto the existing set contents. This is the same as when working with queries and sets.

The Set clause is optional. If no Set clause is included with the Expand Set statement, the output listing of related objects is displayed.

Dump and Output Clauses

You can specify a general output format for listing the expanded information to a file and for output. This is done with the Dump clause:

```
[dump "SEPARATOR_STR"] [output FILENAME]
```

SEPARATOR_STR is a character or character string that should appear between the field values. It can be a tab, comma, semicolon, carriage return, etc. If you do not specify a separator string value, a space is used.

FILENAME identifies a file where the print output is to be stored.

The Dump clause specifies that you do not want to print the leading field name (a space) and that you want to separate the field names with the separator string you provide.

Separator strings can make the output more readable. If many of the business object have similar field values, using tabs as separators will make the values appear in columns.

The Output clause prints the expanded information to a file that you specify (FILENAME).

Tcl Clause

Use the Tcl clause after the dump clause, if used, and before the output clause to return the results in Tcl list format. This facilitates the parsing of output from MQL select commands within Tcl code since the built-in list handling features of Tcl are used directly on the results. For more information, see [Tcl Format Select Output](#) in Chapter 1.

Terse Clause

You can specify the terse clause so that object IDs are returned instead of type, name, and revision. This is done with the Terse clause. For example, the following statement returns a list of object IDs for objects connected to the specified Part:

```
expand set Part "35735" A terse;
```

Viewing Set Definitions

You can view the definition of a set using the Print Set statement. This statement enables you to view all the clauses used to define the set name:

```
print set NAME [SELECT] [dump [SEPARATOR_STR]]  
[recordseparator SEPARATOR_STR] [tcl] [output FILENAME];
```

NAME is the name of the set you want to view.

SELECT specifies the fields you want to print.

SEPARATOR_STR specifies the character or character string that will be used to separate field values from a single business object.

FILENAME outputs the results of the statement to an external file rather than displaying the information on your output terminal.

When you enter this command, MQL displays the business objects (if any) that make up the set. For example, to see the definition for the set named “Seat Components,” you would enter:

```
print set "Seat Components";
```

Remember that names are case sensitive and must be spelled using the same mixture of uppercase and lowercase letters you are unable to access sets created in earlier versions of eMatrix, see your Business Administrator.

If the set name is not found, an error message will result. If that occurs, use the List Set statement to check for the presence and spelling of the set name.

The Print Set statement uses four optional Print statement clauses:

- Select clause
- Dump clause
- Recordseparator clause
- Tcl clause
- Output clause

Each clause is described in the sections that follow.

Select Clause

The Select clause of the Print Set statement enables you to obtain more information than just the type name or revision of the object.

To first examine the general list of field names, use this statement:

```
print businessobject selectable;
```

The selectables for sets are the same as for business objects. Refer to [Select Statements](#) in Chapter 34 for this list.

The Selectable clause is similar to using the ellipsis button in the graphical applications—it provides a list from which to choose.

The Select clause enables you to list all the field names whose values you want to print.

```
select [+] FIELD_NAME [ [FIELD_NAME] ... ]
```

When this clause is included in the Print Set statement, the values of these fields are printed for each business object contained within the set. For example, assume you have four objects within a set named Components. You can see the names and descriptions of each object with the following statement:

```
print set Components select name description;
```

Dump Clause

You can specify a general output format with the Dump clause. The Dump clause specifies that you do not want to print the leading field name and that you want to separate the field names with a separator string that you provide:

```
dump [ "SEPARATOR_STR" ]
```

SEPARATOR_STR is a character or character string that you want to appear between the field values. It can be a tab, comma, semicolon, carriage return, etc. If you do not specify a separator string value, the default value of a comma is used. If tabs or carriage returns are used, they must be enclosed in double quotes ("").

When using the Dump clause, all the field values are printed on a single line unless a carriage return is the separator string. This is very useful when you are processing many business objects in a single print statement. Since the contents of one business object are contained on each line, you can easily use another program to process the print output. When the program reads a carriage return, it knows it has finished with the current business object.

For example, assume you entered the following statement:

```
print set Components select name description  
attribute[ "Date Shipped" ] dump;
```

Separator strings can make the output more readable. If many of the business objects have similar field values, using tabs as separators makes the values appear in columns.

Recordseparator Clause

The Recordseparator clause of the Print Set statement allows you to define which character or character string you want to appear between the dumped output of each object when the Print command requests information about multiple objects.

```
recordseparator [ "SEPARATOR_STR" ]
```

SEPARATOR_STR is a character or character string that replaces the end-of-line (\n) ordinarily inserted at the end of each object's dumped output. It can be a tab, comma, semicolon, etc. If tabs are used, they must be enclosed in double quotes ("").

Tcl Clause

Use the Tcl clause after the dump clause, if used, and before the output clause to return the results in Tcl list format. This facilitates the parsing of output from MQL select commands within Tcl code since the built-in list handling features of Tcl are used directly on the results. For more information, see [Tcl Format Select Output](#) in Chapter 1.

Output Clause

The Output clause of the Print Set statement provides the full file specification and path to be used for storing the Print Set statement output. For example, to store the above information in an external file called components.lst, you might write a statement similar to:

```
print set Components select name description attribute["Date Shipped"] dump output  
$ORDERHOME/shipping/components.lst;
```

Expressions on Sets

Certain kinds of Expressions are applicable to collections of business objects (as can be obtained from Sets and Queries), returning a single answer for the entire collection. These Expressions are formed by using one of several keywords. They are:

count
sum
maximum
minimum
average
median
standarddeviation (stddev or stdDeviation)
correlation (cor)

All but the last of these keywords is expected to be followed by one expression, its *argument*, that applies to business objects. The last one, correlation, needs to be followed by two such expressions. Alternative spellings are indicated in parentheses. For each keyword, you can use all lowercase, all uppercase, or first character uppercase followed by all lowercase.

Count

The count keyword takes as its argument a where-clause expression that evaluates to TRUE or FALSE. It returns an integer that is the number of items in the collection that satisfy the argument. A simple example of such an expression is “count TRUE”, which evaluates to the number of objects in a collection of business objects.

For example, the following statement returns 37, indicating the number of objects of Type ‘Body Panel’ in the database:

```
eval expr 'count TRUE' on temp query bus 'Body Panel' * *;
```

The following statement returns the number of objects in the set “NewBooks” whose cost is between 10 and 50:

```
eval expr 'count ( attribute[cost] >= 10 AND attribute[cost] <= 50 )' on set NewBooks;
```

To do the same, but exclude children’s books, you might use the following statement:

```
eval expr 'count ( attribute[cost] >= 10 AND attribute[cost] <= 50 LESS attribute[Reading Level] == Child )' on set NewBooks;
```

Sum

Sum returns a real number that represents a total of all the values of the specified attribute for all business objects in the collection. For example, the following statement returns the

total of the values of the “Amount Due Employee” attribute for all business objects in the saved query “Expense Reports”:

```
eval expr 'sum attribute[Amount Due Employee]' on query Expense Reports;
```

The following statement returns the ratio of total price to total cost for all objects in the set “Components”:

```
eval expression '( sum attribute[price] ) / ( sum attribute[cost] )' on set Components;
```

Maximum, Minimum, Average

Maximum returns a real number that represents the single largest value for the specified attribute for all business objects in the collection. For example, the following expression checks the value contained in the “diameter” attribute of each business object in the set “o-rings,” and returns whichever value is the highest:

```
eval expr 'maximum attribute[diameter]' on set "o-rings";
```

Minimum returns a real number that represents the single smallest value for the specified attribute for all business objects in the collection.

```
eval expr 'minimum attribute[diameter]' on set "o-rings";
```

Average returns a real number that represents the average of all values for the specified attribute for all business objects in the collection.

```
eval expr 'average attribute[diameter]' on set "o-rings";
```

Median

Median returns a real number that represents the middle number of all values for the specified attribute for all business objects in the collection.

For example, the following shows the values, listed in numerical order, for the attribute Actual Weight for seven business objects that comprise the set FprSet:

7	15	19	25	26	31	35
			↑			

Since the middle number of seven numbers is the fourth number, the median in this case is 25. That is the value returned for the following statement:

```
eval expr 'median attribute[Actual Weight]' on set FprSet;
```

Standard Deviation

Standard deviation, generally used in statistical analysis, tells how closely data conforms to the mean in a set of data. In eMatrix, it can be used to compare the values of business object attributes. The returned value is a real number.

For example, if you know the average age of all employees, you might want to know how many people are close to that age. The standard deviation will tell you, on average, how much the ages of the group differ from the mean. If the standard deviation is a small number, it could indicate that most of the people are close to the average age. If the standard deviation is a large number, it could indicate that there is a broader spread of ages.

The following example performs a standard deviation on the age attribute of all persons in the set Employees:

```
eval expr 'stddev attribute[age]' on set Employees;
```

Correlation

Correlation, generally used in statistical analysis, is a direct measure of the relationship between variables. It can be used in eMatrix to determine the relationship between attributes of an object or group of objects. The returned value (the correlation coefficient) is a real number between -1 and +1.

- If the returned value is between 0 and +1 (positive correlation), it indicates that an increase in the value of one attribute results in an increase in the value of the other (or vice-versa).
- If the returned value is between 0 and -1 (negative correlation), it indicates that an increase in the value of one attribute results in a decrease in the value of the other (or vice-versa).
- A returned value of 0 represents no relationship.

For example, the following expression can be used to check how well cost correlates with price for all objects of Type “tire frame”.

```
eval expr 'cor attribute[Cost] attribute[Price]' on temp query  
bus 'tire frame' * *;
```

Deleting a Set

If you decide that a set is no longer desired, you can delete it using the Delete Set statement:

```
delete set NAME;
```

NAME is the name of the set to be deleted.

When this statement is processed, eMatrix searches the list of existing sets. If the name is found, that set is deleted. If the name is not found, an error message will result.

For example, assume you have a set named “Auto Repairs” that you no longer need. To delete this set from your area, you would enter the following statement:

```
delete set "Auto Repairs";
```

When a set is deleted, there is no effect on the business objects. Remember that grouping business objects together in a set is not the same as connecting them. While connecting business objects makes a global *link* between the objects, sets provide only a local linkage that has no value outside the local user’s context.

Working With Queries

Query Overview	786
Performing a Temporary Query.....	787
Defining a Saved Query	789
Businessobject Clause	790
Hidden Clause	791
Owner Clause	791
Vault Clause	791
Expandtype Clause.....	791
Where Clause	792
Using Select Clauses in Queries	801
Using Fromset and Tosest Selectables.....	805
Query Strategies.....	808
Using Select Fields	808
Using Wildcard Characters in a Where Clause	810
Searching based on lengthy string fields	811
Modeling Considerations	811
Searching on Date/Time	811
Using Parentheses	812
Using const for reserved words	812
Expressions on Queries.....	813
Using the Escape Character	817
To Enable Escaping.....	817
How It Works	818
Using Escaping With Tcl.....	819
Exceptions	819
Modifying a Query Definition.....	820
Evaluating Queries	821
Deleting a Query	823

Query Overview

A *query* is a search on the database for objects that meet the specified criteria. The query is formulated by an individual and, in MQL, it must be saved for subsequent evaluation. A user has access only to queries created during a session under her or his own context. It is then run or *evaluated* and eMatrix finds the objects that fit the query specification. The found objects are displayed in eMatrix or listed in MQL. If the found objects are often needed as a group, they can be saved in a set which can be loaded at any time during an eMatrix or MQL session under the same context.

There are two steps to working with queries:

- *Define* either a saved query that you want to use again, or a temporary query to be used only once for a quick search.
- *Evaluate* the query. The query is processed and any found objects can be put into a set.

Temporary queries allow you to perform a quick search for objects you need only once. In this case you don't have to first save the query itself as an object. For example, you might want to modify a particular object that is named HC-4....., but you have forgotten its full name or capitalization. You could perform a temporary search (without saving the actual query) using "HC-4*" to find all objects that have a name beginning with the letters "HC-4". From the resulting list, you could enter the correct name in your modify statement.

Saved queries allow you to find, for example, all drawings created for a particular project. You can save the results of the query in a set. As the project proceeds, you can also name and save the query itself to use again to update the set contents. Or you might want to repeatedly search for any objects having a particular attribute or range of attributes. Saved queries provide a way of finding all the objects that contain the desired information.

Performing a Temporary Query

You can perform a temporary query that is not first named or saved within the MQL database. In other words, a query can be evaluated without first adding it to the database as an object. The syntax is as follows:

```
temporary query businessobject TYPE NAME REV  
[!expandtype]  
[vault VAULTNAME]  
[owner USERNAME]  
[limit VALUE]  
[querytrigger]  
[where]  
[select]  
[dump "SEPARATOR_STR"]  
[recordseparator "SEPARATOR_STR"]  
[tcl]  
[output FILENAME];
```

For example, to find all business objects in a database use the following:

```
temporary query bus * * *;
```

To find all Assemblies and Assembly subtype objects use:

```
temporary query businessobject Assembly * * expandtype;  
Or  
temporary query businessobject * * * expandtype where type==Assembly;
```

The owner, vault, and where clauses can be used with or without businessobject. For example, you could find all business objects owned by user cslewis, or all business objects of type Part owned by user cslewis:

```
temporary query owner cslewis;  
temporary query bus Part * * owner cslewis;
```

Use the limit clause to control the number of items returned in the search. The system will stop searching after it has reached the specified number of items.

Use the tcl clause after the dump clause, if used, and before the output clause to return the results in Tcl list format. This facilitates the parsing of output from MQL select commands within Tcl code since the built-in list handling features of Tcl are used directly on the results. For more information, see [Tcl Format Select Output](#) in Chapter 1.

Include the querytrigger clause when you want a program named ValidateQuery to be executed. Even with triggers turned off, when querytrigger is included in a query command, the ValidateQuery program gets run. Refer to [Validate Query Trigger](#) in Chapter 20 for more information.

temp query select output is the same as print bus or set select, unless the dump keyword is used. Note the following examples:

Example 1. Temp query without dump keyword:

```
MQL<7>temp query bus Note *e* * select owner;
businessobject Note BingTest 1
    owner = creator
businessobject Note CapTest 1
    owner = creator
businessobject Note attrtest 1
    owner = creator
businessobject Note attrtest1 1
    owner = creator
businessobject Note attrtest2 1
    owner = creator
```

Example 2. Temp query with dump keyword:

```
MQL<8>temp query bus Note *e* * select owner dump;
Note,BingTest,1,creator
Note,CapTest,1,creator
Note,attrtest,1,creator
Note,attrtest1,1,creator
Note,attrtest2,1,creator
```

Example 3. Print set select without dump keyword:

```
MQL<11>print set seltestset select owner;
set seltestset
    member businessobject Note BingTest 1
        owner = creator
    member businessobject Note CapTest 1
        owner = creator
    member businessobject Note attrtest 1
        owner = creator
    member businessobject Note attrtest1 1
        owner = creator
    member businessobject Note attrtest2 1
        owner = creator
```

Example 4. Print set select with dump keyword:

```
MQL<12>print set seltestset select owner dump;
creator
creator
creator
creator
creator
```

Defining a Saved Query

To define a saved query from within MQL, use the Add Query statement:

```
add query NAME {ITEM} ;
```

NAME is the name of the query you are defining. The query name cannot include asterisks.

ITEM specifies the characteristics to search for.

When assigning a name to the saved query, you cannot have the same name for two queries. If you use the name again, an error message will result. However, several different users could use the same name for different queries, since queries are local to the context of individual users.

For example, several users could each have a query called Current Project. But the contents of each Current Project query may differ from user to user depending on the individual needs of each person. As the user adds business objects regarding Current Project, contents may change also. If you change context from one user to another, evaluating the Current Project query will most likely produce different results.

{Item} defines what you are searching for. You can include any or all of the following:

businessobject TYPE_PATTERN PATTERN REVISION_PATTERN
[! not]hidden
owner PATTERN
vault PATTERN
[! not]expandtype
where QUERY_EXPR

None of these clauses is required. The default is an asterisk (*), indicating that the query will find all business objects in the database.

Most of these clauses use PATTERN rather than NAME values. This offers greater flexibility in specifying possible name values. Patterns enable you to use wildcard characters and to list multiple values when specifying a name.

The most commonly used wildcard character in eMatrix queries is the asterisk (*). If only an asterisk is used for the PATTERN, all definitions for that field will be searched. When an asterisk is inserted into a name, it acts as a substitute for a group of letters. This group may contain many letters or none. For example, if you specify a business object name as Ca*, you might find objects named Catalogue93, CadDrawingA49, CaseLog, Ca668, etc. eMatrix searches for all objects that begin with the letters “Ca” and lists any that are found. If you enter a value of dr*t, you will find all objects whose names begin with “dr” and end in “t.”

In this sample query definition below, eMatrix will search for all objects that start with the letters A, B, and C. Each of the other clauses uses only an asterisk for a value. eMatrix will

search to include all vaults and all owners. To restrict the search further, you could include specific values in those clauses.

```
add query "Name Search"  
businessobject * A*,B*,C* *  
vault *  
owner *;
```

The first and last asterisks (*) in the businessobject clause indicate that all types and revisions should be included.

The following sections explain how each Add Query clause is used to filter out and locate desired business objects.

Businessobject Clause

The Businessobject clause of the Add Query statement searches for business objects with a particular or similar name.

```
businessobject TYPE_PATTERN PATTERN REVISION_PATTERN
```

TYPE_PATTERN is a value that translates into one or more business object types.

PATTERN is a value that represents a business object name.

REVISION_PATTERN is a value that translates into one or more revision designators.

For example, the following is a valid Businessobject clause:

```
businessobject Customer Tre*,How* *
```

The first value is an actual object type name: Customer. The second value uses wildcard characters, multiple values, and character strings. A user might search for a customer named Howard Trevor. But the user is unsure of the name spelling and does not know if the customer is stored as Howard or Trevor. By specifying the object name with this pattern, eMatrix searches for all object names that begin with the letters “Tre” or “How.” If any are found that are of type Customer, they are listed. The final asterisk indicates that all revisions are allowed.

When listing multiple values as part of a pattern, you cannot have spaces within the pattern unless the spaces are enclosed within quotation marks. If you accidentally include spaces, eMatrix may read the value as the next part of the object specification. For example, the following clause would produce false values:

```
businessobject Customer Tre*, How* *
```

When eMatrix processes this clause, Customer is again used for the object type. However, rather than searching for names that begin with “Tre” or “How,” eMatrix searches only for objects whose names begin with “Tre”. How* is interpreted as the revision pattern, not part of the name pattern.

For a complete listing of all defined business objects regardless of their exact specification, you can simply insert an asterisk into each pattern of the Businessobject clauses:

```
businessobject * * *
```

Since this clause could produce a large number of objects, it is usually desirable to restrict the query searches in some way. Rather than use an asterisk for each pattern, you may want to use an asterisk in only one or two of the required patterns.

The goal is to provide enough information to filter out unwanted objects. However, you do not want to make your search too narrow or you might miss an important object.

Hidden Clause

You can specify that the new query is “hidden” so that it does not appear in the chooser in eMatrix. Users who are aware of the hidden query’s existence can enter its name manually where appropriate. Hidden objects are accessible through MQL.

Owner Clause

The Owner clause of the Add Query statement searches for business objects that are owned by a particular eMatrix user:

```
owner NAME_PATTERN
```

NAME_PATTERN is the owner(s) you are searching for.

The Owner clause uses wildcard characters in a way similar to the Businessobject clause.

You can also use multiple values to define the pattern. You can search for both the first and last name of an owner. For example, the following clause specifies all objects whose owner names begin with pat or thur:

```
add query "Object Owner Search" owner pat*,thur*;
```

This statement might find objects created by owners patricia, patty, and thurston.

Vault Clause

The Vault clause of the Add Query statement searches for business objects that are in a particular or similar vault:

```
vault PATTERN
```

PATTERN is the vault(s) you are searching for.

The Vault clause uses wildcard characters in a way similar to the Businessobject clause.

For example, the following query definition requests all business objects that reside in the “Vehicle Project” vault:

```
add query "Vault Search"
businessobject * * *
vault "Vehicle Project";
```

Vaults defined as remote (for a loosely coupled database) must be explicitly listed in order to be searched. Using an asterisk in the Vault clause searches only local and Foreign (Adaplet) vaults.

Expandtype Clause

The Expandtype clause of the Add Query statement is used to find all the specified type hierarchy of types. This is the default.

For example, a type of Frame Assembly may have derived types of Handle and Guard. To limit the search to objects with a type of Frame only, use !expandtype:

```
add query "Frame Assembly"
businessobject Frame * *
!expandtype;
```

Where Clause

The Where clause of the Add Query clause is the most powerful and the most complicated. It searches for selectable business object properties. This involves examining each object for the presence of the property and checking to see if it meets the search criteria. If it does, the object is included in the query results; otherwise, it is ignored.

While the Where clause can test for equality, it can test for many other characteristics as well. The syntax for the Where clause details the different ways that you can specify the search criteria, using familiar forms of expression:

```
where "QUERY_EXPR"  
Or  
where 'QUERY_EXPR'
```

QUERY_EXPR is the search criteria to be used.

In general, the syntax for a query expression is in one of the following forms:

(QUERY_EXPR)
ARITHM_EXPR RELATIONAL_OP QUERY_EXPR
BOOLEAN_EXPR
If-Then-Else
Substring

Each form is described further in the sections that follow.

Rules of thumb for creating where clauses in MQL queries:

- Always enclose the entirety of the where clause in SINGLE quotes. If the expression is not enclosed, eMatrix will not read it as a single value. Most query expressions contain multiple values and spaces. Therefore quotes are necessary to determine the boundaries of the expression.
- Always include a space on either side of each arithmetic operator (+, -, *, /) to correctly separate it from the operands.
- Strings in square brackets can have spaces—the square brackets delimit them, but any other strings with spaces should be enclosed in DOUBLE quotes.
- When using where clauses with expand bus, you must always insert select bus or select rel before a where clause. See [SELECT_BO and SELECT_REL Clauses](#) in Chapter 35 for details.

Query Expression (QUERY_EXPR)

This form simply means that a query expression can be contained within parentheses. Although not required, parentheses can add to readability and are useful when writing complex expressions. Parentheses can be used to group subclauses of the where clause, but not as start/end delimiters.

Comparative Expressions

This form of a Boolean expression considers comparisons such as greater than, less than, and equality. You have two arithmetic expressions and a relational operator:

```
ARITHM_EXPR RELATIONAL_OP ARITHM_EXPR
```

`ARITHM_EXPR` is an arithmetic expression that yields a single value. This value may be numeric, a character string, a date, or a Boolean. Arithmetic expressions are further described below.

`RELATIONAL_OP` is a relational operator. Relational operators and the comparison they perform are summarized in the table *Relational Operators (RELATIONAL_OP)*.

When writing comparative expressions, you can have any mixture of arithmetic expressions. Since all arithmetic expressions yield a single value, they are interchangeable as long as they are of the same type. You cannot mix different value types in the same comparison expression. If you try to mix types, an error message will result.

You cannot compare values of different types because some operators do not work with some values. For example, testing for an uppercase or lowercase match does not make sense if you are working with numeric values. Even when you have values of the same type, you must be sure to use a relational operator that is appropriate for the values being compared. If the operator is incorrect for the values being compared, an error message will result.

Arithmetic Expressions (ARITHM_EXPR)

Use the following syntax:

```
ARITHM_EXPR BINARY_ARITHMETIC_OP ARITHM_EXPR
```

`ARITHM_EXPR` can be a selectable field name that yields a numeric value, an arithmetic operand (value), or another arithmetic expression. While arithmetic expressions include all data types, arithmetic expressions apply only to Integer or Real value types.

`BINARY_ARITHMETIC_OP` is one of four arithmetic operators:

- Plus sign (+) for addition
- Minus sign (-) for subtraction
- Asterisk (*) for multiplication
- Slash (/) for division

Arithmetic expressions can be written three ways:

<code>FIELD_NAME</code>	eMatrix uses the value contained within the named field for each object. This field name and its notation can be found by using the Print Businessobject Selectable statement (see <i>Select Statements</i> in Chapter 34.)
<code>VALUE</code>	eMatrix uses the value that you provide.
<code>ARITH_EXPR</code>	eMatrix performs one or more arithmetic operations to arrive at a single numeric value.

These forms allow you to write comparative expressions such as:

attribute[Units] eq Inches	Compares the values of the Units attribute to see if it is equal to Inches. If it is, the object is included with the query output.
"attribute[Product Cost]" > "attribute[Maximum Cost]"	Compares the value of the Product Cost attribute with the value of the Maximum Cost attribute. If the Product Cost exceeds the Maximum Cost, the object will be included in the query output.
("attribute[Parts In Stock]" - 10) < ("attribute[Parts Needed]" + 5)	Evaluates the results of each arithmetic expression and checks to see if the first result is less than the second. If it is, the object is included in the query output.

In the last comparative expression, all three forms of an arithmetic expression are used. This expression compares the results of two arithmetic expressions. One value in each arithmetic expression is a field name (Parts In Stock or Parts Needed) and one is a value supplied by you (10 or 5). Before the comparison can take place, each arithmetic expression must be evaluated and reduced to a single value. Then the two values can be compared.

*Be sure to include a space on either side of each arithmetic operator (+, -, *, /) to correctly separate it from the operands.*

String Concatenation

When using the plus (+) operator, if the left-hand operand is a string, the system attempts to obtain a string from the right-hand operand. If successful, the “+” operation returns a string obtained by concatenating the two strings.

```
eval expr '( "actual=" + attribute[Actual Weight]) + (",
target=" + attribute[Target Weight])' on bus 'Body Panel'
610210 0;
```

This statement produces the following results:

```
actual=29.5, target=29.0
```

Using Dates in Expressions

The following calculations can be performed on dates within an expression:

- subtract two dates, obtaining a number of seconds
- add or subtract a number (of seconds) to/from a date
- use the string MX_CURRENT_TIME for the current date/time

For example, the following could be used to determine how old an object is (in hours):

```
evaluate expr '(MX_CURRENT_TIME - state[Released].actual) / 3600' on bus 'Body Panel'
610210 0
```

The following returns the average age of all objects in the set M6000-panels (in hours):

```
evaluate expr 'average ( (MX_CURRENT_TIME - state[Released].actual) / 3600)' on set
M6000-panels;
```

Relational Operators (RELATIONAL_OP)

Relational operators can be used to compare values of all data types unless specified otherwise.

Operator	Operator Name	Function
<code>==</code> <code>eq</code> <code>EQ</code>	is equal to	The first value must be equal to the second value. When comparing characters, uppercase and lowercase are not equivalent.
<code>!=</code> <code>neq</code> <code>NEQ</code>	is not equal to	The first value must not match the second value. When comparing characters, uppercase and lowercase are not equivalent.
<code><</code> <code>lt</code> <code>LT</code>	is less than	The first value must be less than the second value. This comparison is not normally used with Boolean data types. When comparing dates, an older date has a lesser value.
<code>></code> <code>gt</code> <code>GT</code>	is greater than	The first value must be greater than the second value. This comparison is not normally used with Boolean data types. When comparing dates, the more recent date has the greater value.
<code><=</code> <code>le</code> <code>LE</code>	is less than or equal to	The first value must be equal to or less than the second value. This operator is not used with Boolean data types.
<code>>=</code> <code>ge</code> <code>GE</code>	is greater than or equal to	The first value must be greater than or equal to the second value. This operator is not used with Boolean data types
<code>~~</code> <code>smatch</code> <code>SMATCH</code>	string match	The general pattern of the first value must match the general pattern of the second value. The value can be included anywhere in the string. With this operator, character case is ignored so that "redone" is considered a match for "RED*."
<code>!~~</code> <code>nsmatch</code> <code>NSMATCH</code>	not string match	The general pattern of the first value must not match the general pattern of the second value. The value can be included anywhere in the string. With this operator, character case is ignored. For example, a first value of "Red Robbin" and a second value of "rE* rO*" would result in a FALSE comparison since the two are considered a match regardless of the difference in uppercase and lowercase characters.
<code>~=</code> <code>match</code> <code>MATCH</code>	case-sensitive match	The pattern of the first value must match the pattern of the second value. The value can be included anywhere in the string. This includes testing for uppercase and lowercase characters. For example, "Red Robbin" is not a sensitive match for the pattern value "re* ro*" because the uppercase "R" values will not match the pattern's lowercase specification.

<code>~=</code>	case-sensitive	The pattern of the first value must match the pattern of the second value. The value can be included anywhere in the string. This includes
<code>match</code>	<code>match</code>	testing for uppercase and lowercase characters. For example, "Red Robbin" is not a sensitive match for the pattern value "re* ro*" because the uppercase "R" values will not match the pattern's lowercase specification.
<code>MATCH</code>		

Operator	Operator Name	Function
smatchlong	string match for long data	Contains the specified value, in either the lxStringTables or the lxDescriptionTables. The search is NOT case sensitive. To find all objects with the word "Language" in the attribute Comments, and to ensure that both tables are checked, use attribute[Comments] smatchlong "language" in the Where clause. Since the query is not case sensitive, the query will find objects with "language", "Language" or "LANGUAGE" in the Comments field.
nsmatchlong	not string match for long data	Does not contain the specified value, in either the lxStringTables or the lxDescriptionTables. The search is NOT case sensitive. The "n" is for not match. To find all objects that do not contain the word "language" in the attribute Comments, enter "language" for the value. Because the query is not case sensitive, it would not find objects with the word written as "LANGUAGE" or "Language" in the Comments attribute, as well as those that did not contain the word at all.

To find objects where a particular property is non-blank, use a double asterisk. For example, if you want be sure that all objects in the result of your query contain a description, use:

```
where '(description=="**")';
```

Matchlist Expressions

Expressions can use two keywords, matchlist and smatchlist, which enable you to specify a list of strings on the right-side of these keywords. These work the same as the match and smatch keywords except that an additional operand is used as a separator character for the list of strings. The format is:

matchlist 'STRING_LIST' ['SEPARATOR_CHAR']
smatchlist 'STRING_LIST' ['SEPARATOR_CHAR']

where

STRING_LIST is the list of strings to be compared

SEPARATOR_CHAR defines the character that separates the list of strings. If no separator character is given, the first right-hand operand is treated exactly as match and smatch treat it, that is, as one string.

Following are some examples:

```
temp query bus Errata * * where "current matchlist 'Open,Test' ','";
temp query bus Errata * * where "attribute[Priority] matchlist '0,1,2' ','";
temp query bus Errata * * where "attribute[Notes] smatchlist '*Yin*,*Williams*' ,'"
```

The last query will find all Errata in which the names "Yin" or "Williams" are included in the Notes section, and the search will be case-insensitive.

Note that in the above examples, a comma is specified as the separator character in the second right-hand operand. The first right-hand operand is treated as a list of strings delimited by this separator character.

*In matchlist and smatchlist expressions, the following cannot be used as a separator character: * . \ [that is: asterisk, period, or backslash]*

If either left or right operand is a select clause, the result of evaluating the operand is a list of strings. In all other cases, the evaluation of the operand just gives a single string. `match`, `smatch`, `matchlist`, and `smatchlist` are evaluated by a pair-wise comparison of the two lists. If any comparison yields true, then the result is true. Otherwise, it is false.

Note that when a pair-wise comparison is performed, asterisks and question marks are treated as wildcard characters only in the string(s) from the right-hand operand. Asterisks and question marks from the left-hand operand are treated as a literal string, and not as wildcard characters.

Searching based on lengthy string fields

The eMatrix/Oracle database stores most string attribute values in the `IxStringTable` for the object's vault. However, `IxStringTable` cannot hold more than 254 bytes of data. When a string attribute's value is larger than this limit, the data is stored in the descriptions table (`IxDscriptionTable`), and a pointer to this table is placed in the `IxStringTable`.

When performing an "includes" search (using match operators: `match`, `match case`, `not match`, `not match case`) on string attribute values, eMatrix searches on both `IxDscription` and `IxString` tables when the attribute involved is of type "multiline." Also, if you use the equal operators (`==`, `!=`) and give a string of more than 254 bytes to be equal to, eMatrix checks the values in the `IxDscriptionTable` only.

To search on description or other string attribute values for given text, and to force the search of both tables, you can use the "long" match operators. These operators can be used in any expression (including the `where` clause entry screen) but are not shown as options in the query dialog's Where Pattern in either the desktop or Web version of eMatrix.

Alternatively, you can include the `.value` syntax in the `where` clause, as shown below:

```
attribute[LongString].value ~ "matchstring"
```

Boolean Expressions: BOOLEAN_EXPR

This form of query expression means that the query expression can be either a single arithmetic expression or a selectable business object property that yields an arithmetic expression. For example, assume you want to find a list of honor students. The criteria for honor roll may be described as:

```
where 'attribute["Grade Point Average"]>=3.8'
```

When this clause is processed, eMatrix looks for all objects that have this attribute and includes the value of "Grade Point Average." If the attribute is not found within the object definition or if the attribute value returns false when the `Where` clause is evaluated, the object is excluded from the query output. To include a selectable object property in a `Where` clause, you must use its proper name and notation.

You can obtain this by using the `Print Businessobject Selectable` statement (see [Select Statements](#) in Chapter 34.) This statement prints a list of all field names that can be used

and indicates how they must be written. Refer also to the selectables listed in the *Select Expressions* appendix in the *eMatrix Navigator Guide*.

A Boolean expression contains one or more comparisons. These comparisons return a value of TRUE, FALSE or UNKNOWN. For example, UNKNOWN might be returned as a string by a string attribute. It could also be returned by a Program, since Programs can be invoked in Expressions and return a string.

Boolean expressions, whose values can be True, False, or Unknown, should not be confused with Boolean type attributes, whose values can be only True or False.

As with TRUE and FALSE, mixed case (Unknown) and lowercase (unknown) are allowed.

The syntax for a Boolean expression uses two basic forms:

QUERY_EXPR BINARY_BOOLEAN_OP QUERY_EXPR
UNARY_BOOLEAN_OP QUERY_EXPR

The first form of the Boolean expression assumes you have two Boolean values whose relationship you want to compare. While there are only three Boolean operators, there are multiple ways that these operators can be specified. Refer to the table below.

When specifying a Boolean value in this form, follow the same syntax rules as for specifying a query expression. You can create long lists of Boolean expressions. For example, you could write Where clauses such as:

where 'type==Student && attribute["Grade Point Average"]==4.0'
where 'current=="Initial Testing" && attribute[compound]==steel && attribute[weight] < 2.5'

When MQL processes these clauses, it obtains the Boolean values for each field name and then evaluates the Boolean relationship. If the results of the evaluation are true, the object is included in the query output. If false, it is excluded.

Boolean “and” can be represented: AND, and , &&.

Boolean “or” can be represented: OR, or , | |.

Boolean “unknown” can be represented: UNKNOWN, unknown.

The following table shows the results for Boolean relationships:

Boolean Relationship	Results
TRUE and TRUE	TRUE
TRUE and FALSE	FALSE
TRUE and UNKNOWN	UNKNOWN
FALSE and UNKNOWN	FALSE
TRUE or TRUE	TRUE
TRUE or FALSE	TRUE
TRUE or UNKNOWN	TRUE
FALSE or UNKNOWN	UNKNOWN

The second form of the Boolean expression assumes that you have a single Boolean value. This value (represented by QUERY_EXP) can be operated upon to yield yet another Boolean value. The unary Boolean operator is the NOT operator. This operator causes eMatrix to use the opposite value of the current Boolean value. For example, with the expression NOT(True), the final value of the Boolean expression would be False. The unary operator has three different notations:

```
NOT  
not  
!
```

!UNKNOWN yields a result of UNKNOWN

The one you use is a matter of preference.

Complex Boolean Expression

With the addition of comparative expressions, query expressions can become very complicated. You can specify any amount of search criteria. Only if all the criteria is met will the object be included.

The more criteria you list, the more difficult it is to maintain readability. Remember that you can use parentheses to help readability.

When MQL encounters a complex query expression, it uses the following rules to evaluate it:

1. All arithmetic expressions are evaluated from left to right and innermost parentheses to outermost.
2. All comparative expressions are evaluated from left to right and innermost parentheses to outermost.
3. All AND operations are evaluated from left to right and innermost parentheses to outermost.
4. All OR operations are evaluated from left to right and innermost parentheses to outermost.

For example, you might use the following query to find all Drawings connected to Assembly types that have not been released and have connections from Markups.

```
print query relationship;  
query relationship  
    business * * *  
    vault *  
    owner *  
where '(type==Drawing)  
    && (relationship[Drawing].to.type==Assembly)  
    && (current==Released)  
    && (relationship[Markup].from.type==Markup)' ;
```

Note that the type can be specified as part of the business object or in a boolean expression.

If-Then-Else

If-then-else logic is available for Expressions. The syntax is:

```
if EXPRESSION1 then EXPRESSION2 else EXPRESSION3
```

The EXPRESSION1 term must evaluate to TRUE, FALSE, or UNKNOWN.

If the EXPRESSION1 term evaluates to UNKNOWN, it is treated as TRUE.

The if-then-else expression returns the result of evaluating EXPRESSION2 or EXPRESSION3 depending on whether or not EXPRESSION1 is TRUE or FALSE.

Note that only one of EXPRESSION2 or EXPRESSION3 is evaluated. So if the expressions have side-effects (which can happen since expressions can run programs), these effects will not occur unless the expression is evaluated.

```
eval expr ' if (attribute[Actual Weight] > attribute[Target Weight]) then ("OVER") else ("OK")' on bus 'Body Panel' 610210 0;
```

Substring

The substring operator works within an expression to provide the ability to get a part of a string; the syntax is:

```
substring FIRST_CHAR LAST_CHAR EXPRESSION
```

The substring operator works as follows:

- The FIRST_CHAR and LAST_CHAR terms must evaluate to numbers that are positive or negative, and whose absolute value is between 1 and the number of characters in the string returned by EXPRESSION.
- The numbers returned by these terms indicate a character in the string returned by EXPRESSION.
- The characters are counted so that '1' refers to the first character. A negative number indicates the character found by counting in the reverse direction. So '-1' refers to the last character.
- The substring operator returns the part of the string returned by EXPRESSION consisting of the characters from the FIRST_CHAR character to the LAST_CHAR character, inclusive.
- If FIRST_CHAR evaluates to a character that is after the character indicated by LAST_CHAR, an empty string is returned.

To obtain the last 4 characters of a 10-character phone number, use:

```
eval expression 'substring -1 -4 attribute[Phone Number]' on bus Vendor 'XYZ Co.' 0;
```

Using Select Clauses in Queries

The purpose of select expressions is to obtain or use information related to a particular business object. In a query, the select expression value is used to qualify the search criteria (in a Where clause) by comparing it with another (given) value.

Obtainable information includes not only attribute values and other business object data, but also administrative object information, such as the governing policy, vault, and so on. The key property of a select expression is that it can access information *related* to an object.

In all cases, the expression is processed from the context of a starting object. In a query, the starting points are business objects that meet other selection criteria (vault, type, and so on). The phrase *starting point* is used because the select mechanism in eMatrix actually uses the same concept of navigation from one object to another that makes the rest of the system so flexible. This is possible because most information in eMatrix is actually represented internally by a small object and not by a text string or numeric value as it appears to the user.

These internal objects are all linked in much the same way business objects are connected by relationships. The links can be traversed to travel from one object to another (navigation). The presence of these links is indicated in the select expression notation by a period.

The remaining sections in this chapter provide examples and information about select expressions for queries in MQL. See the *Select Expressions* appendix in the *eMatrix Navigator Guide* for more information.

Definitions

- A select clause is single-valued if `print bus T N R select <clause> dump` outputs exactly one line of text.
- A select clause is multi-valued if `print bus T N R select <clause> dump` outputs multiple lines of text.
- A select clause is NULL if `print bus T N R select <clause> dump` outputs zero lines of text.

Using the Format File Dump Clause

Assume you have the following four objects:

- `Assembly A` has a file checked into format Assembly
- `Assembly W` has file checked into format Word
- `Assembly AW` has a file checked into each format: Assembly and Word
- `Assembly NONE` has no files checked in.

Single-valued—Each single-valued select clause produces a single field of output. Select clauses that do NOT allow square brackets are all single valued. Selecting attributes are a special case; they allow 0 or 1 value to be output according to whether the attribute is present on the business object. In keeping with the desire to have the number of outputs equal the number of select clauses, attribute selects output an empty field in the case where the attribute does not exist on the business object.

However, other clauses that allow square brackets are format, state, relationship, to, from, revisions, history, and method. These are used in two ways:

To check for existence. For example:

```
relationship[BOM];  
format[ASCII];  
format[ASCII].hasfile.
```

In these cases, the number of outputs is not ambiguous, and a true or false value is always returned.

To get subfields. For example:

```
relationship[BOM].to.name;  
from[ ].to.name format[ASCII].file.
```

These may represent zero, one or many pieces of data, depending on the number of relationships, formats, files, etc. that are possessed by the business object, so it is not possible to guarantee that the number of outputs will equal the number of select clauses. Therefore, selecting subfields will produce output fields only for data that is actually present. They do NOT output empty fields to represent the absence of data.

If you enter the single-valued Print Businessobject statement for Assembly A, as follows:

```
print bus Assembly A '' select format[ ].file dump;
```

the results indicate a Word document is included in the object Assembly A:

```
andersen:D:\test\Monitor FSP.doc:
```

If you enter the single-valued Print Businessobject statement for Assembly W

```
print bus Assembly W '' select format[ ].file dump;
```

the results indicate a Word document is included in the object Assembly W:

```
andersen:d:\test\Monitor FSP.doc:
```

Multi-valued—If you enter the multi-valued Print Businessobject statement for Assembly AW:

```
print bus Assembly AW '' select format[ ].file dump;
```

the results indicate two Word documents are included in the object Assembly AW, separated by a comma:

```
andersen:d:\test\select.txt,  
andersen:d:\test\Monitor FSP.doc:
```

NULL-valued—If you enter the NULL valued Print Businessobject statement for Assembly NONE:

```
print bus Assembly NONE '' select format[ ].file dump;
```

the results are NULL.

Using the Format Hasfile Dump Clause

If you use the `format[].hasfile dump` clause instead of the `format[].file dump` clause, the value of TRUE is returned for each document included in the object.

If you enter the Print Businessobject statement with the `hasfile` clause for Assembly A

```
print bus Assembly A '' select format[ ].hasfile dump;
```

the TRUE response indicates that there is a document included in the object Assembly A, without its file details.

If you enter the Print Businessobject statement with the `hasfile` clause for Assembly W

```
print bus Assembly W '' select format[ ].hasfile dump;
```

the TRUE response indicates that there is a document included in the object Assembly W, without its file details.

If you enter the Print Businessobject statement with the hasfile clause for Assembly AW

```
print bus Assembly AW '' select format[ ].hasfile dump;
```

the TRUE , TRUE response indicates there are two documents included in the object Assembly AW, without their file details. One TRUE is displayed for each document.

If you enter the Print Businessobject statement with the hasfile clause for Assembly NONE

```
print bus Assembly NONE '' select format[ ].hasfile dump;
```

the NULL response indicates that there are no documents included in the object Assembly NONE.

NULL clauses

Select clauses that are found to be NULL have special handling for the *equal* and *not equal* logical operators: ==, ~~ , ~=, !=, !~, and !=~.

- A NULL select clause is NEVER equal (==, ~~ , ~=) to anything
- A NULL select clause is ALWAYS not equal (!=, !~, !=~) to everything.

So, for our example, we have:

```
temp query bus Assembly * '' where 'format[Assembly].hasfile==TRUE' ;
```

results in:

```
Assembly A  
Assembly AW
```

and then:

```
temp query bus Assembly * '' where 'format[Assembly].hasfile != TRUE' ;
```

results in:

```
Assembly NONE  
Assembly W
```

Multi_valued Select Clauses

Multi-valued select clauses are handled as a string of OR's. That is, each of the multiple values is used separately to evaluate the boolean expression. If any one of these single-valued comparisons are TRUE, the whole multi-valued comparison is considered TRUE.

For example, consider these objects:

- Assembly A contains an ASCII file:

```
format.file = d:\doc\select.txt
```
- Assembly AW contains an ASCII file and a Word document

```
format.file = d:\doc\select.txt  
format.file = d:\doc\specification.doc
```
- Assembly W contains a Word document

```
format.file = d:\doc\specification.doc
```

- Assembly DELETED contains no files.
- Assembly NONE contains no files.

If you enter:

```
temp query bus Assembly * * where ' "format.file" MATCH "*.doc" ';
```

it results in:

Assembly AW (multi-valued, and 2nd one is a match)
Assembly W

If you enter:

```
temp query bus Assembly * * where ' "format.file" MATCH "*.txt" ';
```

it results in:

Assembly A
Assembly AW (multi-valued, and 1st one is a match)

Note that using NMATCH will also pick up the objects with no files. If you enter:

```
temp query bus Assembly * * where ' "format.file" NMATCH "*.txt" ';
```

it results in:

Assembly AW (multi-valued, and NMATCH is TRUE
for the 2nd one)
Assembly W (singlevalued, and NMATCH is TRUE)
Assembly DELETED (NULL, so NMATCH is always TRUE)
Assembly NONE (NULL, so NMATCH is always TRUE)

Using Fromset and Tosest Selectables

Two selectables are available for business objects, `fromset[]` and `tosest[]`, that make it possible to obtain information about the relationships from or to a given object that have an object of a given set at the other end. In particular, they can be used in `where` clauses of queries as a way to specify that an object be returned only if it is at the “to” or “from” end of a relationship of a certain type having an object of a given set at the other end. The use of these keywords in this manner solves problems of functionality and performance that are difficult, if even possible, to solve any other way.

Suppose the following query is run:

```
temp query bus RMS * * where ((current==Approved) &&
(attribute[Preferred_Material_Flag]~~'TRUE') &&
(to[Owning_Security_Group-Technical_Standard].from.name~~'*~BABY CARE~*') &&
(to[Owning_Security_Group-Technical_Standard].from.name~~'*~NORTH AMERICA~*'));
```

The way the system runs such a query is that it would first find all objects that matched this query:

```
temp query bus RMS * * where ((current==Approved) &&
(attribute[Preferred_Material_Flag]~~'TRUE'));
```

Then the system would test the truth of the remaining clauses against each object found from the first query. This method has the potential of being extremely slow. There may be a large number of objects returned that have to be checked against the remaining clauses, and few of them might test true. Furthermore, the work needed to test the remaining clauses can be very intensive. All the to-relationships of an object must be obtained and then the name of the object at the other end must be tested until a match is found. Since the

objects at the other ends can live in different vaults, a single join cannot accomplish this goal, so multiple SQL commands are needed.

These performance issues can be avoided by using the `toset` [] selectable. Two MQL statements are required instead of one, so you must perform the query in a program. The two statements would be:

```
temp query bus * **~BABY CARE~*, *~NORTH AMERICA~* " * into set t1;
temp query bus RMS * * where ((current==Approved) &&
(attribute[Preferred_Material_Flag]~~'TRUE') &&
(toset[t1,Owning_Security_Group-Technical_Standard]==True);
```

The first statement finds all objects that satisfy the conditions that the “objects at the other end” needed to satisfy. In this example, these conditions have been replaced by a clause that uses `toset` []. These selectables, `toset` [] (and `fromset` []), must include brackets that contain a set name followed, optionally, by a relationship type name. Multiple relationship names can be given, each separated by a comma from the previous one. In the above example, `toset` [] returns True if there is a relationship from an object in the set `t1` of type `Owning_Security_Group-Technical_Standard` to the object being tested. If no relationship type is specified, the query returns True as long as some relationship exists between such objects, regardless of type. The `fromset` [] selectable works the same as `toset` [] except that the ends are reversed—the relationship must be *to* an object in `t1` and *from* the object being tested.

With `toset` and `fromset` selectables, the values “True” and “False” are case sensitive and always appear in title case (initial capital letter followed by lower case letters). In queries with these selectables, you must type “True” or “False” using this case to get a valid result.

What distinguishes this query from the single-statement query is that the `toset` condition can be included with the other conditions when objects are gathered so that eMatrix sees only objects that satisfy all the conditions of the query. As long as the first query does not put too many objects into set `t1`, this query should have much better performance.

The `fromset` and `toset` selectables can be followed by additional selectables, similar to the way that the selectables `from` and `to` work. In such cases, the selectable that follows is evaluated against each relationship that satisfies the condition indicated by the `fromset` or `toset` selectable. For example,

`fromset[t1,assembly,component].name` returns a list of the names of the relationships of type `assembly` or `component` from a given object to an object in set `t1`.

Additionally, these selectables can be used to express conditions that cannot be expressed in a single query. Using the example above, the following conditions:

```
(to[Owning_Security_Group-Technical_Standard].from.name~~'*~BABY CARE~*' )
```

and

```
(to[Owning_Security_Group-Technical_Standard].from.name~~'*~NORTH AMERICA~*' )
```

each express that a given object be the to-object in a relationship of type `Owning_Security_Group-Technical_Standard` where the object at the other end satisfies a given condition. These two conditions can be true even if no one object at the other end satisfies both conditions. But it may be intended that the two conditions be satisfied by the same object. This query does not express that, and no single query can. However, `toset` [] can be used to express this condition by making sure that the set in question contains only objects that satisfy both conditions. This goal is accomplished by replacing the statement that created the set by this statement:

```
temp query bus * **~BABY CARE~*" * where "name match '*~NORTH AMERICA~*' " into set t1;
```

Sets in eMatrix are workspace objects and are shared by anyone using the same login name. If two people (or applications) are logged in with the same user name, they may overwrite each other's sets if the same names (t1, t2, etc.) are used. To avoid this, developers should wrap the creation of the set and the final query inside a transaction boundary. This will keep the set from becoming visible to other sessions until it is no longer needed.

Query Strategies

The following section contains hints concerning how to include Where clauses within a query.

- The key consideration for optimizing query performance is to use as many indexed fields as possible (see [Using Select Fields](#) for a list). This limits the number of business objects returned to the client in order to perform the non-indexed evaluation. It's also important to limit the use of wildcards for type, name, and revision, and to specify a search vault.
- You must include a space before and after the operator in all Where expressions. In complicated expressions, particularly those that use "!" or "not", use parentheses to clearly state your intent. For example, in the following, the "!" is applied to the entire clause:

```
!relationship[Categorize] || relationship[Categorize].from.id == 43482.46832.5291.38424  
To apply the "!" to only the portion before the OR, change it to:
```

```
(!relationship[Categorize]) || relationship[Categorize].from.id == 4448.10921.4769.5768
```

- To find objects where a particular property is blank, use a double quote (no space). For example, if you want find objects that do not contain a description, use (description ~~~"). For Boolean attributes, searching on "" results in finding objects where the value for the attribute is False, even if the default is True. (Boolean attributes, are either True or False, as opposed to Boolean expressions, which may be True, False or Unknown).
- To find objects where a particular property is non-blank, use a double asterisk. For example, if you want be sure that all objects in the result of your query contain a description, use (description ~~~**").
- When entering a date as a value in the Where clause box, eMatrix assumes the time is 12:00 AM unless a specific time is specified as part of the query.
- Do not use any word that can be a selectable for a business object or connection as a value in the Where clause of a query because it will be evaluated as a select clause rather than being used literally. Refer to the Select Expressions Appendix in the *eMatrix Navigator Guide* for a complete list of business object selectables.

Other issues and strategies are described in the sections that follow.

Using Select Fields

Type name and revision fields in a query are used to limit the number of objects returned to the client. The additional criteria in the where clause are used to filter the objects displayed. Criteria that limits the number of objects to be filtered are said to be "indexed." The key consideration for optimizing query performance is to use as many indexed fields as possible.

The following selectables when present in a query where clause, are treated as indexed fields by the query optimizer.

Indexed Field	Meaning
Name	object name
Type	object type
Owner	object owner
Policy	governing policy of object
Format	format of files checked into object
Originated	date object was created
Modified	date object was last modified
Current	current state of object
attribute[]	value of attribute on object
to[].attribute[]	value of attribute on relationship connected to object
from[].attribute[]	value of attribute on relationship connected from object
relationship[].attribute[]	value of attribute on relationship connected to or from object
format.file.store	store containing files checked into object
format.file.location	location containing files checked into object
search[]	result of full text search

You can use indexed fields with any relational operator (==, !=, <> etc.) applied to a constant value which may include wildcards. For example,

```
name ~= 'A*B'
```

returns all objects whose name begins with “A” and ends with “B”. Note that use of select fields in where clauses that are not in the list above will likely result in non-optimal queries since non-indexed fields have to be evaluated on the eMatrix client or the Collaboration Server.

Non-Indexed Clauses

Any selectables not on the above list are non-indexed when used in a where clause. For example, the following are not indexed:

```
description
revision
grantor (and grantee, granteesignature)
state[]
revisions[]
previous (and next, first, last)
```

Note that revision is not indexed when included in the where clause, but is indexed when included in the business object specification part of the query.

Non-indexed selectables cannot be built into the SQL commands that get objects and connections from the database server, so these clauses are not used to limit the number of

objects that are retrieved from the server. Qualifying these clauses is very sensitive to the number of objects retrieved, since for each such object, additional SQL calls have to be constructed to retrieve the values of these fields, and data has to be stored in the clients memory. Given these realities:

NEVER execute a query that has only non-indexed fields.

ALWAYS make sure that a query containing non-indexed fields also has some indexed fields to limit the number of objects retrieved.

Here are a few examples of common queries that are bound to be slow. Note that in all of them, the type (PART) is specified, but a production database can have huge numbers of PARTs, so the type specification is not very limiting:

```
PART * * where 'description ~~ "*a word*"'  
PART * * where 'state[StateName].satisfied == TRUE'  
PART * * where 'revision == last'
```

In the above queries, it would be best to include an attribute that substantially limits the number of PARTs that the system would have to examine.

The number of objects retrieved by the indexed fields is the biggest factor in query performance and client memory requirements. The length (in characters) of the where clause or number of expressions within the where clause are immaterial. Actually, a longer where clause is desirable if much of it is comprised of indexed fields that limit the number of objects retrieved:

```
PART * * where ' (description ~~ "*a word*") AND  
(attribute[Keyword] ~~ "abc*") AND (attribute[Keyword] ~~  
"123*") '
```

Indexed Attributes

If your queries consistently include the same set of attribute criteria, ask your system administrator about creating an Index for your attributes. Query performance could be improved by as much as 100 times.

Using Wildcard Characters in a Where Clause

When used with the Equal and Not Equal operators (==, !=) in the where clause, the system treats the wildcard characters “?” and “*” as literal characters. Do not use these characters when querying using the Equal or Not Equal operators. On the other hand, eMatrix interprets * and ? as wildcards when used with the four Match operators and when used in the Type, Name, or Revision fields of the query.

For example, if you type the following query:

```
temp query bus * A*B *
```

eMatrix returns all objects that start with A and end with B. However, if use the equality operator in the where clause as follows:

```
temp query bus * * * where name == A*B
```

eMatrix looks for the literal A*B as the entire object name rather than treating the * as a wildcard.

Searching based on lengthy string fields

The eMatrix/Oracle database stores most string attribute values in the lxStringTable for the object's vault. However, lxStringTable cannot hold more than 254 bytes of data. When a string attribute's value is larger than this limit, the data is stored in the descriptions table (lxDescriptionTable), and a pointer to this table is placed in the lxStringTable.

When performing an “includes” search (using match operators: match, match case, not match, not match case) on string attribute values, eMatrix searches on both lxDescription and lxString tables when the attribute involved is of type “multiline.” Also, if you use the equal operators (==, !=) and give a string of more than 254 bytes to be equal to, eMatrix checks the values in the lxDescriptionTable only.

To search on description or other string attribute values for given text, and to force the search of both tables, you can use the “long” match operators. These operators may be used in any expression (including the where clause entry screen) but are not offered in the query dialog in either the desktop or web version of eMatrix. Refer to [Relational Operators \(RELATIONAL_OP\)](#) for more information.

Alternatively, you can include the .value syntax in the where clause, as shown below:

```
attribute[LongString].value ~~ "matchstring"
```

For more information about .value keyword, refer to the Select appendix in the *eMatrix Navigator Guide*.

Modeling Considerations

It may be appropriate to include some consideration of query performance when designing your data model—in particular, when defining a new type.

For example, here are three modeling alternatives that minimize the need for queries on descriptions. All three can easily be maintained using the ModifyDescription trigger on the type:

- Add an attribute Synopsis, which holds the first 100 characters of the description field.
- Add an attribute Keywords, and require values to be specified at create time.
- Write the description to a text file, check it into the object, and replace uses of 'description ==' by 'search[] =='.

Searching on Date/Time

In order to include Date/Time in your query, you must use the format defined for your system initialization file. eMatrix provides six different formats for the display and entry of dates and times. Each can be modified by adding lines to the .ini file or the startup scripts. See *Configuring Date and Time Formats* in the *eMatrix Installation Guide* for details.

If your Date/Time query does not conform to the expected format, you will receive an error message similar to the following:

```
Invalid date/time format 'July 22. '02'.
Allowed formats are:
[day] mon dom, yr4 h12:min[:sec] [mer] [tz]
[day] mon dom, yr4
moy/dom[/yr2] h12:min[:sec] [mer]
moy/dom[/yr2]
```

The following table explains the tokens used in Date formats:

Token	Meaning
day	day of the week (mon, tue, wed,...)
DAY	day of the week, not abbreviated
mon	month (jan, feb, mar,...)
MON	month, not abbreviated
tz	time zone (edt, cdt, pdt,...)
TZ	time zone, not abbreviated
mer	time meridian (am, pm, or blank for 24 hour time)
sec	seconds, 0 - 59
min	minutes, 0 - 59
h12	hour in 12 hour format, "mer" will be non-blank
h24	hour in 24 hour format, "mer" will be blank
yr2	abbreviated year (96, 97, 98,...)
yr4	full year (1996, 1997, 1998..)
dom	day of month (1, 2, 3,..., 31)
doy	day of year (1, 2, 3,..., 365)
moy	month of year (1, 2, 3,..., 12)

Using Parentheses

In complicated expressions, particularly those that use ! or not, use parentheses to clearly state your intent. For example, in the following, the ! is applied to the entire clause:

```
!relationship[Categorize] || relationship[Categorize].from.id=="43482.46832.5291.38424
```

To apply the ! to only the portion before the OR (||), change it to:

```
(!relationship[Categorize]) || relationship[Categorize].from.id==4448.10921.47699.5768
```

Using const for reserved words

Reserved words such as keywords and select expressions must be afforded special consideration in exact equal (==) eMatrix expressions. For example, the following statement might be written to find business objects where the value of the attribute 'Regression' is 'first'.

```
temp query bus * * *
where 'attribute[Regression]==first';
```

But because ‘first’ is a select keyword that returns the first revision of a business object and is evaluated as such, the result of the evaluation — rather than the literal word ‘first’— is compared with the attribute. (For a list of keywords and their meanings, see the *Select Expressions* appendix in the *eMatrix Navigator Guide*.)

For this type of situation, use `const` to indicate that whatever follows should not be evaluated. For example:

```
temp query bus * * *
where 'attribute[Regression]==const"first"';
```

`Const` has three possible forms: all uppercase, all lowercase, and initial letter capitalized followed by all lowercase. No space can appear after the word `const`. It must be followed by a quote (double or single, depending on the syntax of the rest of the statement). Almost any character can appear within the quotes, with the exception of backslash and pound sign. The characters between the initial and closing quotes remain unevaluated.

If your implementations are using JSP/Tcl to compose a `where` clause dynamically (that is, using a variable to construct the `where` clause), the `const` syntax must be used because the value you pass in could be an eMatrix keyword. If this happens, the `where` clause will not return an error, but you will get unexpected results.

Here are some examples of queries that will NOT return correctly without using `const`:

```
attribute[Some Attribute]==‘first’
attribute[Some Attribute]==‘FIRST’
attribute[Some Attribute]==“Current”
attribute[Some Attribute]==“owner”
description==“Current”
description==“policy”
```

However, the following will return correctly:

```
attribute[Some Attribute]==‘my first order’
attribute[Some Attribute]~='*first*'
```

Expressions on Queries

Certain kinds of Expressions are applicable to collections of business objects (as can be obtained from Sets and Queries), returning a single answer for the entire collection. These Expressions are formed by using one of several keywords. They are:

count
sum
maximum
minimum
average
median
standarddeviation (stddev or stddeviation)
correlation (cor)

All but the last of these keywords is expected to be followed by one expression, its *argument*, that applies to business objects. The last one, correlation, needs to be followed by two such expressions. Alternative spellings are indicated in parentheses. For each keyword, you can use all lowercase, all uppercase, or first character uppercase followed by all lowercase.

Count

The count keyword takes as its argument a where-clause expression that evaluates to TRUE or FALSE. It returns an integer that is the number of items in the collection that satisfy the argument. A simple example of such an expression is “count TRUE”, which evaluates to the number of objects in a collection of business objects.

For example, the following statement returns 37, indicating the number of objects of Type ‘Body Panel’ in the database:

```
eval expr 'count TRUE' on temp query bus 'Body Panel' * *;
```

The following statement returns the number of objects in the set “NewBooks” whose cost is between 10 and 50:

```
eval expr 'count ( attribute[cost]>=10 AND attribute[cost]<=50 )' on set NewBooks;
```

To do the same, but exclude children’s books, you might use the following statement:

```
eval expr 'count ( attribute[cost]>=10 AND attribute[cost]<=50 LESS attribute[Reading Level]==Child )' on set NewBooks;
```

Sum

Sum returns a real number that represents a total of all the values of the specified attribute for all business objects in the collection. For example, the following statement returns the total of the values of the “Amount Due Employee” attribute for all business objects in the saved query “Expense Reports”:

```
eval expr 'sum attribute[Amount Due Employee]' on query Expense Reports;
```

The following statement returns the ratio of total price to total cost for all objects in the set “Components”:

```
eval expression '( sum attribute[price] ) / ( sum attribute[cost] )' on set Components;
```

Maximum, Minimum, Average

Maximum returns a real number that represents the single largest value for the specified attribute for all business objects in the collection. For example, the following expression checks the value contained in the “diameter” attribute of each business object in the set “o-rings,” and returns whichever value is the highest:

```
eval expr 'maximum attribute[diameter]' on set "o-rings";
```

Minimum returns a real number that represents the single smallest value for the specified attribute for all business objects in the collection.

```
eval expr 'minimum attribute[diameter]' on set "o-rings";
```

Average returns a real number that represents the average of all values for the specified attribute for all business objects in the collection.

```
eval expr 'average attribute[diameter]' on set "o-rings";
```

Median

Median returns a real number that represents the middle number of all values for the specified attribute for all business objects in the collection.

For example, the following shows the values, listed in numerical order, for the attribute Actual Weight for seven business objects that comprise the set FprSet:

7	15	19	25	26	31	35
↑						

Since the middle number of seven numbers is the fourth number, the median in this case is 25. That is the value returned for the following statement:

```
eval expr 'median attribute[Actual Weight]' on set FprSet;
```

Standard Deviation

Standard deviation, generally used in statistical analysis, tells how closely data conforms to the mean in a set of data. In eMatrix, it can be used to compare the values of business object attributes. The returned value is a real number.

For example, if you know the average age of all employees, you might want to know how many people are close to that age. The standard deviation will tell you, on average, how much the ages of the group differ from the mean. If the standard deviation is a small number, it could indicate that most of the people are close to the average age. If the standard deviation is a large number, it could indicate that there is a broader spread of ages.

The following example performs a standard deviation on the age attribute of all persons in the set Employees:

```
eval expr 'stddev attribute[age]' on set Employees;
```

Correlation

Correlation, generally used in statistical analysis, is a direct measure of the relationship between variables. It can be used in eMatrix to determine the relationship between attributes of an object or group of objects. The returned value (the correlation coefficient) is a real number between -1 and +1.

- If the returned value is between 0 and +1 (positive correlation), it indicates that an increase in the value of one attribute results in an increase in the value of the other (or vice-versa).
- If the returned value is between 0 and -1 (negative correlation), it indicates that an increase in the value of one attribute results in a decrease in the value of the other (or vice-versa).
- A returned value of 0 represents no relationship.

For example, the following expression can be used to check how well cost correlates with price for all objects of Type “tire frame”.

```
eval expr 'cor attribute[Cost] attribute[Price]' on temp query bus 'tire frame' * *;
```

Using the Escape Character

The backslash (\) character can be used in MQL commands or expressions to escape any other character in a variety of contexts. In particular, MQL users can create attribute values that contain both single and double quotes, and search for objects with an attribute value of this sort.

When escaping is enabled, the character that follows a backslash loses any special meaning it might have in the particular context.

You can specify that a backslash (\) should be treated as an escape character for any other character when used in:

- MQL commands
- expressions, such as used in the following:
 - where clauses (including queries, expand businessobject command, cues, tips, and filters)
 - object tip definitions
 - table column definitions
 - evaluate expression command
 - expression access
- output of a range program
- program arguments, such as appear in these contexts:
 - execute program command
 - execute businessobject command
 - triggers
 - various places in wizard definitions
- combo and list boxes in wizards

This can be specified globally or on a case-by-case basis (except for wizard components).

To Enable Escaping

To enable the use of the backslash as an escape character globally, Business Administrators can use the following MQL command:

```
set escape on;
```

Other commands available are:

```
set escape off;
```

```
print escape;
```

- When `escape` is set to `on`, a backslash will always act as an escape character.
- Use `print escape` to check whether it is enabled or not.
- Use `set escape off` to disable it.

The escape status is determined the first time an application is started. When setting escape, all newly started applications will use the setting, but any applications that were already started will not, including the session that made the setting. Business Administrators should decide which way to set escape and set it once.

If you want to avoid enabling the escape character globally, you can enable it on an as needed basis, prefixing each relevant string to be parsed with the keyword `escape` plus a space. When processed on the command line, these extra characters are first stripped off and the remainder will be processed in the usual way.

When more than one expression is specified in a command or definition, (for example, in `tip` definitions and `expand bus` where commands) you can escape one and not the other by including the escape clause in only the expression that needs it.

How It Works

When an MQL command is processed by the system, the system first breaks the command into parts that are called tokens. When escape processing is on for the command (whether by the global or ad-hoc setting), it affects the breakdown of the command into tokens. An expression embedded in a command is treated as one token, from the point of view of the command parser. Prefixing a command with the word “escape” affects only the processing of the command into tokens. For example the following would not work as required:

```
temp query bus * * * where "attribute[height]==5' ";
```

Without using the escape mechanism and backslashes, this `where` clause would be processed as:

```
attribute[height]==
```

The text after the equal sign would be ignored, since the second quotation mark (before the 5) seems to indicate the end of the `where` clause (`where` clauses must be enclosed in single or double quotes). To correctly search for objects that have a height of five feet, you would use:

```
escape temp query bus * * * where "attribute[height]==\\"5'\\"";
```

With the backslashes and escape processing turned on, the `where` clause of this command is correctly processed as:

```
attribute[height]=="5'
```

In some cases, however, escape processing is needed for both parsing the command into tokens and also for parsing the `where` clause for evaluation. The `where` clause parsing is done after the command is parsed as a second pass. The rules for this parsing are somewhat different, as expressions have their own syntax and special keywords. For example, to find all objects that have a height of 5'6” is tricky, since the value itself contains both single and double quotes and must also be surrounded by quotes. In this case you would need to escape process the `where` clause (and include the escape character) in order to make it work as a `where` clause, as follows:

```
escape attribute[height]=='5\'6'
```

To perform the query, you would then need to use escape processing at the command level as well as at the expression level. You could use the following:

```
escape temp query bus * * * where "escape attribute[height]=='5\\\'6\\\'';"
```

Notice that you must escape the backslash so that one would be in place after the initial escape processing of the command, for the second pass that escape processes the `where` clause. A triple backslash is required if the character used to enclose the expression is included in the expression itself, such as:

```
escape temp query bus * * * where 'escape attribute[height]==\\'5\\\\'6\"\\'';'
```

In summary, escapes are needed when you want to place a value within a string that will be parsed; in order to have the string parsed as one token, you need to place quotes around it. Without escape processing, this only works if the string does not itself contain the quote character. With escape processing, you can use such quotes as a delimiter safely if you modify the string by following these simple rules in this order:

1. Escape each escape character in the string.
2. Escape each quote character (of the same type — single or double) in the string.

Expressions and program arguments that start with “escape” will be treated similarly, as will the output of a range program.

Using Escaping With Tcl

When using Tcl you can enable escaping (either globally or within the command) to put any kind of string into the database or pass such a string as an argument to an MQL program regardless of which characters it contains and without the need for the user to worry about backslashes at all—except for those necessitated in order to set Tcl variables. In general, by using Tcl you can avoid the need to add quotes around tokens for the purpose of making the MQL command processor treat them correctly. That then removes the need to use an escape character with such quotes. When an MQL command is executed in Tcl, it is Tcl that initially breaks the command into tokens. When eMatrix receives these tokens from Tcl, it attempts, as best as possible, to put them together into a command string so that the MQL command processor will treat each of the tokens received from Tcl as a single token. So if the token contains a space or a quote character or any of a number of other characters, internally eMatrix will put quotes around it and then put it together with the other tokens to create an MQL command string that is parsed as any normal MQL command is parsed. When escape processing is on, eMatrix will also escape the contents of the string properly so that it will be interpreted as one token by the MQL command processor.

So, in Tcl programs, if you put values into a variable and then use the variable to set attribute values, you should enable escape (either globally or within the command), and eMatrix will add the necessary backslashes to the processed commands. For example:

```
tcl;
set myvar { my string with many weird characters such as ' ;#" - but no squiggly brace }
mql escape modify businessobject type name rev MyAttr $myvar
```

If you needed to use a squiggly brace inside the squiggly braces of the `set` command, you would have to backslash it — to satisfy Tcl's demands, but not MQL's.

Exceptions

One exception to escape processing is that it cannot be used for characters that are part of a keyword in an expression. Keywords are such things as `match`, `attribute`, `ge`, `==`, and `AND`. They have a special meaning within expressions and backslashing their characters is neither necessary nor will it work properly.

Use of special characters in administrative object names must be avoided. Even escaping special characters in this case does not work as shown in the example below (the name of the attribute is “My]Attr”):

```
escape print bus MyType MyName MyRev select attribute[My\]Attr];
```

There is also one instance where a backslash does not change the function of the following character. In an MQL command, if a line with a comment ends with a backslash, the comment does not continue to the next line.

Modifying a Query Definition

You change the search criteria for an existing query by using the Modify Query statement:

```
modify query NAME {ITEM}:
```

NAME is the name of the query you want to modify.

ITEM is the type of modification you want to make. With the Modify Query statement, you can use these modification clauses to change a query:

businessobject TYPE_PATTERN PATTERN REVISION_PATTERN
owner PATTERN
vault PATTERN
[! not]expandtype
where PATTERN

These clauses are the same clauses that are used to define the initial query. When making modifications, you simply substitute new values for the old.

Although the Modify Query statement allows you to use any combination of search criteria, no other modifications can be made. To change the query name or remove the query entirely, you must use the Delete Query statement (see [Deleting a Query](#)) and/or create a new query.

For example, assume you have a query named “Product Comparison” with the following definition:

```
query "Product Comparison"
  businessobject *
  revision *
  type Perfume
  vault "Perfume Formulas"
  owner channel, taylor;
```

To this query, you want to add another owner for the search criteria. To make the change, you would write a Modify Query statement similar to the following:

```
modify query "Product Comparison"
  owner channel, taylor, cody;
```

This alters the query so that it now appears as:

```
query "Product Comparison"
  businessobject *
  revision *
  vault "Perfume Formulas"
  owner channel, taylor, cody
```

Evaluating Queries

Once a query is defined, you need to evaluate it, using the Evaluate Query statement, to find the information.

```
evaluate query NAME;
```

When a query is evaluated, all business objects that meet the search criteria are displayed in the window or listed on your screen. If you want to save this collection of objects, you can assign a set name to it and reference the set name when you want to view the collection. (Refer also to *Set Defined* in Chapter 37.)

Saving query results as a set can be useful when you have a changing environment. Even when you use the same query, it is possible that you would get different results if the objects are undergoing change. Therefore, to save query results for a later time, you should place them in a set.

Use the Evaluate Query statement to process the query and optionally save the results of a query in a set. This statement contains the following optional clauses:

```
into set SET_NAME  
onto set SET_NAME  
over set SET_NAME into|onto  
querytrigger
```

QUERY_NAME is the name of the query to be used.

SET_NAME is the name to be assigned to the collection of objects created by the query.

INTO form:	If the named set exists, eMatrix clears the set and places the results of the current query into the set. If the named set does not exist, eMatrix creates the set and places the results into it.
ONTO form:	If the named set exists, eMatrix adds the results of the current query to the set contents. If the named set does not exist, eMatrix creates the set and places the results into it.
OVER form:	eMatrix performs a find on an existing set.
querytrigger	Executes the program named ValidateQuery, even if triggers are turned off. Refer to <i>Validate Query Trigger</i> in Chapter 20 for more information.

If the INTO or ONTO form of the Evaluate Query statement is used, the found set is not listed on the screen. To view them, you must print the set. If you only evaluate the query and do not save onto or into a set, the found values are listed on the screen. For example:

```
add query sarah where 'type==drawing';  
print query sarah;  
query sarah  
businessobject * * *  
vault *  
owner *  
where 'type==Drawing'  
eval query sarah;  
Drawing test A  
Drawing ttest A
```

```
Drawing 726602 A
Drawing 726601 A
Drawing 726600 A
Drawing 726596 B
Drawing 726595 A
Drawing 726594 A
Drawing 726593 A
Drawing 726592 A
Drawing 726591 C
Drawing 726590 B
Drawing 50234 F
Drawing 50225 D
Drawing 50461 B
Drawing 50023 F
Drawing 50403 B
eval query sarah into set sarah;
print set sarah;
set sarah
member businessobject Drawing test A
member businessobject Drawing ttest A
member businessobject Drawing 726602 A
member businessobject Drawing 726601 A
member businessobject Drawing 726600 A
member businessobject Drawing 726596 B
member businessobject Drawing 726595 A
member businessobject Drawing 726594 A
member businessobject Drawing 726593 A
member businessobject Drawing 726592 A
member businessobject Drawing 726591 C
member businessobject Drawing 726590 B
member businessobject Drawing 50234 F
member businessobject Drawing 50225 D
member businessobject Drawing 50461 B
member businessobject Drawing 50023 F
member businessobject Drawing 50403 B
```

Deleting a Query

If a query is no longer needed, you can delete it using the Delete Query statement:

```
delete query NAME;
```

NAME is the name of the query to be deleted.

When this statement is processed, eMatrix searches the local list of existing queries. If the name is found, that query is deleted. If the name is not found, an error message results.

For example, assume you have a query named “Overdue Invoices” that you no longer need. To delete this query from your area, you would enter the following MQL statement:

```
delete query "Overdue Invoices";
```

After this statement is processed, the query is deleted and you receive the MQL prompt for the next statement.

When a query is deleted, there is no effect on the business objects or on queries performed by other eMatrix users. Queries are local only to the user’s context and are not visible to other eMatrix users.

Working With Filters

Filter Defined.....	826
Creating Filters	827
Active Clause.....	827
Applies to Clause.....	827
Direction Clause	827
Type Clause.....	828
Name Clause.....	828
Revision Clause.....	828
Vault Clause	829
Owner Clause	829
Where Clause	829
Hidden Clause	830
Property Clause	830
Copying and/or Modifying a Filter.....	831
Copying (Cloning) a Filter Definition.....	831
Modifying a Filter	831
Deleting a Filter.....	833

Filter Defined

Filters limit the objects or relationships displayed in eMatrix browsers to those that meet certain conditions previously set by you or your eMatrix Business Administrator. For example, you could create a filter that would display only objects in a certain state (such as Active), and only the relationships connected *toward* each object (not *to and from*). When this filter is turned on, only the objects you needed to perform a specific task would display.

From eMatrix Navigator browsers, filters that limit the number of objects that display can be very useful. Each user can create her/his own filters from eMatrix Navigator (or MQL). However, if your organization wants all users to use a basic set of similar filters consistently, it may be easier to create them in MQL, then copy the code to each user's personal settings (by setting context).

Filters can be defined and managed from within MQL or from the Visuals Manager in eMatrix Navigator. Once the filters are defined, individual users can turn them on and off from the eMatrix browsers as they are needed.

In the eMatrix Navigator browsers, filters display on the Filters tab page within the Visuals Manager window, in the Filter bar and in the View menu.

It is important to note that filters are Personal Settings that are available and activated only when context is set to the person who defined them.

Creating Filters

To define a new filter from within MQL, use the Add Filter statement:

```
add filter NAME [ ADD_ITEM {ADD_ITEM} ];
```

NAME is the name of the filter you are defining.

ADD_ITEM specifies the characteristics you are setting.

When assigning a name to the filter, you cannot have the same name for two filters. If you use the name again, an error message will result. However, several different users could use the same name for different filters. (Remember that filters are local to the context of individual users.)

After assigning a filter name, the next step is to specify the conditions (ADD_ITEM) that each object must meet in order to display when the filter is turned on (activated). The following are Add Filter clauses:

[! in notin not]active
appliesto businessobject relationship
from to both
type TYPE_PATTERN
name PATTERN
revision REVISION_PATTERN
vault PATTERN
owner PATTERN
where QUERY_EXPR
[! not]hidden
property NAME on ADMIN [to ADMIN] [value STRING]

Each clause and the arguments they use are discussed in the sections that follow.

Active Clause

The Active clause is used to indicate that the filter is active or not active (!active). The default is active.

Applies to Clause

The Applies to clause indicates that the filter applies to business objects or relationships.

Direction Clause

The Direction clause indicates the direction of the relationships to which the filter applies: to, from, or both.

Type Clause

The Type clause of the Add filter statement assigns a filter for a particular type of business object or relationship.

```
type TYPE_PATTERN
```

TYPE_PATTERN defines the types for which you are assigning a filter.

The Type clause can include more than one type by using multiple values to define the pattern. The Type clause can also use wildcard characters. For example, the following definition displays a filter to display all customers and other users:

```
add filter "Customers and Other Users" type customer,user;
```

When listing multiple values as part of a pattern, separate each value with a comma and no spaces, or enclose any multi-word type value within quotation marks (e.g., "new customer", user). If you include spaces, eMatrix reads the value as the next part of the filter definition. For example, the following Type clause would produce an error because MQL reads the type as "new" and the next specification for the filter as "customer".

```
type new customer;
```

Name Clause

The Name clause of the Add filter statement assigns the names of objects to include in the filter.

```
name PATTERN
```

PATTERN is the names of objects to include in your filter.

The Name clause can include more than one name by using multiple values to define the pattern. The Name clause can also use wildcard characters. For example, the following definition adds a filter to display all business objects with names that start with "Inter", or include the letters "IBM", or include the words "International Business Machines":

```
add filter "IBM"  
name Inter*, *IBM*, "International Business Machines";
```

When listing multiple values as part of a pattern, separate each value with a comma and no spaces, or enclose any multi-word type value within quotation marks (for example, "International Business Machines"). If you include spaces, eMatrix reads the value as the next part of the filter definition, as described in the Type clause.

Revision Clause

The Revision clause of the Add filter statement assigns a filter for a particular revision of business objects.

```
revision REVISION_PATTERN
```

REVISION_PATTERN defines the revision for which you are assigning a filter.

The Revision clause can include more than one revision value and wildcards as in the other clauses that use patterns. Typically, you might use a wildcard, but would not include more than one or two revisions. Filtering by the latest revision number or all revisions A and B can remove much of the out-dated business objects from view. This allows you to work with only the most current objects.

The Revision clause can also use wildcard characters. For example, the following definition displays a filter to display all parts of revision 1 or 2:

```
add filter "Revised Parts"
type parts
revision 1,2;
```

When listing multiple values as part of a pattern, separate each value with a comma and no spaces, or enclose any multi-word type value within quotation marks (for example, "status new", "status old"). If you include spaces, eMatrix reads the value as the next part of the filter definition.

Vault Clause

The Vault clause of the Add Filter statement assigns a filter for business objects that are in a particular or similar vault:

```
vault PATTERN
```

PATTERN defines the vault(s) for which you are assigning a filter.

The Vault clause can use wildcard characters. For example, the following definition displays a filter for all business objects that reside in the "Vehicle Project" vault:

```
add filter "Vault Search"
appliesto businessobject
vault "Vehicle Project"
owner *;
```

Owner Clause

The Owner clause of the Add Filter statement assigns a filter for business objects that were created by a particular owner:

```
owner PATTERN
```

NAME_PATTERN defines the owner(s) for which you are creating a filter.

The Owner clause can use wildcard characters. You can also use multiple values to define the pattern. You can specify both the first and last name of an owner. For example, the following Owner clause specifies all objects whose owner names begin with pat or thur:

```
add filter "Object Owner" owner pat*,thur*;
```

This statement might provide a filter for objects created by owners patricia, patty, and thurston.

Where Clause

The Where clause of the Add Filter statement is the most powerful and the most complicated. It searches for selectable business object properties. This involves examining each object for the presence of the property and checking to see if it meets the search criteria. If it does, the object is displayed with the filters applied; otherwise, it is ignored.

While the Where clause can test for equality, it can test for many other characteristics as well. The syntax for the Where clause details the different ways that you can apply the filter, using familiar forms of expression:

```
where "QUERY_EXPR"  
Or  
where 'QUERY_EXPR'
```

QUERY_EXPR is the criteria to be used.

For detailed information, see [Where Clause](#) in Chapter 38.

Hidden Clause

You can mark the new filter as “hidden” so that it does not appear in the chooser in eMatrix. Users who are aware of the hidden filter’s existence can enter the name manually where appropriate. Hidden objects are accessible through MQL.

Property Clause

Integrators can assign ad hoc attributes, called Properties, to the filter. Properties allow associations to exist between administrative definitions that aren’t already associated. The property information can include a name, an arbitrary string value, and a reference to another administration object. The property name is always required. The value string and object reference are both optional. The property name can be reused for different object references, that is, the name joined with the object reference must be unique for any object that has properties.

```
add filter NAME property NAME [to ADMINTYPE NAME] [value STRING];
```

For additional information on properties, see *Overview of Administration Properties* in Chapter 24.

Copying and/or Modifying a Filter

You can modify any filter that you own, and copy any filter to your own workspace that exists in any user definition to which you belong. As an alternative to copying definitions, you can use the `set workspace` statement to make your workspace look like that of another user. Business Administrators can change their workspace to that of another user to work with filters that they do not own. See [Setting the Workspace](#) in Chapter 44 for details.

Copying (Cloning) a Filter Definition

After a filter is defined, you can clone the definition with the `Copy Filter` statement. Cloning a filter definition requires Business Administrator privileges, except that you can copy a filter definition to your own context from a group, role or association in which you are defined.

This statement lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy filter SRC_NAME DST_NAME [fromuser USER] [touser USER] [overwrite] {MOD_ITEM}  
{MOD_ITEM};
```

`SRC_NAME` is the name of the filter definition (source) to be copied.

`DST_NAME` is the name of the new definition (destination).

`USER` can be the name of a person, group, role, or association.

`Overwrite` replaces an existing visual in the destination user.

The order of the `fromuser`, `touser` and `overwrite` clauses is irrelevant, but `MOD_ITEMS`, if included, must come last.

`MOD_ITEMS` are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

Modifying a Filter

The `List` statement is used to display a list of all filters that are currently defined. It is useful in confirming the existence or exact name of a filter, since eMatrix is case-sensitive.

```
list filter [user USER_NAME];
```

`USER_NAME` can be the name of any person, group, role, or association. If the `user` modifier is used, the system displays the requested information relating to workspace objects of the person, group, role, or association indicated.

Users can list their own workspace objects or those of any group, role, or association to which they belong. Business Administrators can list the workspace objects of any users.

Use the list of all the existing filters along with the `Print` statement to determine the search criteria you want to change. Use the `Modify Filter` statement to add or remove defining clauses and change the value of clause arguments:

```
modify filter NAME [ITEM {ITEM}];
```

NAME is the name of the filter you want to modify.

ITEM is the type of modification you want to make. With the Modify filter statement, you can use these modification clauses to change a filter:

Modify Filter Clause	Specifies that...
active	The active option is changed to specify that the object is active.
notactive	The active option is changed to specify that the object is not active.
appliesto businessobject relationship	The appliesto option is changed to reflect what the filter now affects.
from to both	The direction of the filter is changed.
type TYPE_PATTERN	The type is changed to the named pattern.
name PATTERN	The name is changed to the named pattern.
revision REVISION_PATTERN	The revision is changed to the named pattern.
vault PATTERN	The vault is changed to the pattern specified.
owner PATTERN	The owner is changed to the pattern specified.
where QUERY_EXPR	The query expression is modified.
hidden	The hidden option is changed to specify that the object is hidden.
nothidden	The hidden option is changed to specify that the object is not hidden.
property NAME [to ADMINTYPE NAME] [value STRING]	The named property is modified.
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.

These clauses are essentially the same ones that are used to define an initial filter except that Add property and Remove property clauses are substituted for the Property clause. When making modifications, you simply substitute new values for the old.

Although the Modify filter statement allows you to use any combination of criteria, no other modifications can be made except the ones listed. To change the filter name or remove the filter entirely, you must use the Delete filter statement and/or create a new filter.

Deleting a Filter

If a filter is no longer needed, you can delete it using the Delete filter statement:

```
delete filter NAME;
```

NAME is the name of the filter to be deleted.

When this statement is processed, eMatrix searches the local list of existing filters. If the name is found, that filter is deleted. If the name is not found, an error message results.

For example, assume you have a filter named “Overdue Invoices” that you no longer need. To delete this filter from your area, you would enter the following MQL statement:

```
delete filter "Overdue Invoices";
```

After this statement is processed, the filter is deleted and you receive the MQL prompt for the next statement.

When a filter is deleted, there is no effect on the business objects or on queries performed by eMatrix. Filters are local only to the user’s context and are not visible to other eMatrix users.

40

Working With Cues

Cues Defined.....	836
Creating a Cue	837
Adding a Cue	837
Active Clause.....	838
Appliesto Clause.....	838
Type Clause.....	838
Name Clause.....	838
Revision Clause.....	838
Color, Font, Highlight, and Linestyle Clauses.....	839
Vault Clause	839
Order Clause	839
Owner Clause	840
Where Clause	840
Hidden Clause	840
Property Clause	840
Copying or Modifying a Cue.....	841
Copying (Cloning) a Cue Definition	841
Modifying a Cue.....	841
Deleting a Cue.....	844

Cues Defined

Cues control the appearance of business objects and relationships inside any eMatrix browser. They make certain objects and relationships stand out visually for the user who created them.

This appearance control is based on conditions that you specify such as attribute value, current state, or lateness. Objects and relationships that meet the criteria may appear in any distinct color, line, or font style.

You can save a cue and not make it active. This allows for multiple or different sets of conditions to be used at different times or as a part of different Views. See *Using View Manager* in the *eMatrix Navigator Guide*. Cues are set using the Visuals Manager and saved as a query in your personal settings. They can be activated from the Visuals Manager Cue tab or from the View menu.

From eMatrix Navigator browsers, cues that highlight the appearance of certain objects can be very useful. Each user can create her/his own cues from eMatrix Navigator (or MQL). However, if your organization wants all users to use a basic set of similar cues consistently, it may be easier to create them in MQL, then copy the code to each user's personal settings (by setting context).

Cues can be defined and managed from within MQL or from the Visuals Manager in eMatrix Navigator. Once the cues are defined, individual users can turn them on and off from the eMatrix browsers as they are needed.

In the eMatrix Navigator browsers, they display on the Cues tab page within the Visuals Manager window and in the View menu.

It is important to note that cues are all Personal Settings that are available and activated only when context is set to the person who defined them.

Creating a Cue

From eMatrix browsers, unique cues can be very useful when interacting with many objects. Each user can create her/his own cues from eMatrix Navigator (or MQL). However, if your organization wants all users to see a basic set of similar cues consistently, it may be easier to create them in MQL, then copy the code to each user's personal settings (by setting context).

Adding a Cue

To create a new cue from within MQL, use the Add Cue statement:

```
add cue NAME [ADD_ITEM {ADD_ITEM}] ;
```

NAME is the name of the cue you are defining. Cue names cannot include asterisks.

ITEM specifies the characteristics you are setting.

When assigning a name to the cue, you cannot have the same name for two cues. If you use the name again, an error message will result. However, several different users could use the same name for different cues. (Remember that cues are local to the context of individual users.)

After assigning a cue name, the next step is to specify cue characteristics (ITEM). The following are Add Cue clauses:

[! in notin not]active
appliesto businessobject relationship all
type TYPE_PATTERN
name PATTERN
revision REVISION_PATTERN
color COLOR
font FONT
highlight COLOR
vault PATTERN
linestyle solid bold dashed dotted
order -1 0 1
owner PATTERN
where QUERY_EXPR
[! not]hidden
property NAME on ADMIN [to ADMIN] [value STRING]

Each clause and the arguments they use are discussed in the sections that follow.

Active Clause

The Active clause is used to indicate that the cue is active or not active (!active). The default is active.

Appliesto Clause

The Appliesto clause indicates that the cue applies to a business object, a relationship, or both (all). Since relationships and objects can have the same attributes, cues must specify to which they should apply.

When defining a cue for an object, you can use color and font to highlight it. When defining a cue for a relationship you can use color and line style, as described below.

Type Clause

The Type clause of the Add Cue statement assigns a cue for a particular type of business object or relationship.

```
type TYPE_PATTERN
```

TYPE_PATTERN defines the types for which you are assigning a cue.

The Type clause can include more than one type by using multiple values to define the pattern. The Type clause can also use wildcard characters.

When listing multiple values as part of a pattern, separate each value with a comma and no spaces, or enclose any multi-word type value within quotation marks. If you include spaces, eMatrix reads the value as the next part of the cue definition.

Name Clause

The Name clause of the Add Cue statement assigns the names of objects to include in the cue.

```
name PATTERN
```

PATTERN is the names of objects to include in your cue.

The Name clause can include more than one name by using multiple values to define the pattern. The Name clause can also use wildcard characters.

When listing multiple values as part of a pattern, separate each value with a comma and no spaces, or enclose any multi-word type value within quotation marks (for example, "International Business Machines"). If you include spaces, eMatrix reads the value as the next part of the cue definition, as described in the Type clause.

Revision Clause

The Revision clause of the Add Cue statement assigns a cue for a particular revision of business objects.

```
revision REVISION_PATTERN
```

REVISION_PATTERN defines the revision for which you are assigning a cue.

The Revision clause can include more than one revision value and wildcards, as in the other clauses that use patterns. Typically, you might use a wildcard, but would not include more than one or two revisions. Adding a cue for the latest revision number can highlight only the most current business objects in a view.

When listing multiple values as part of a pattern, separate each value with a comma and no spaces, or enclose any multi-word type value within quotation marks. If you include spaces, eMatrix reads the value as the next part of the cue definition.

Color, Font, Highlight, and Linestyle Clauses

The Color, Font, Highlight, and Linestyle clauses of the Add Cue statement define the visual characteristics of the objects/relationships. The options for an object are color and font, while the options for a relationship are color and line style.

```
color COLOR
font FONT
highlight COLOR
linestyle solid|bold|dashed|dotted
```

COLOR is the name of the color to display for the object text and/or the relationship arrow. When COLOR is defined for highlight, this applies when the object is highlighted (selected).

You can enter a style of solid, bold, dashed, or dotted for the relationship arrow displayed between objects. This control is available only if you are applying the cue to relationships.

Vault Clause

The Vault clause of the Add Cue statement assigns a cue for business objects that are in a particular or similar vault:

```
vault PATTERN
```

PATTERN defines the vault(s) for which you are assigning a cue.

The Vault clause can use wildcard characters. For example, the following definition displays a cue for all business objects that reside in the “Vehicle Project” vault:

```
add cue "Vault Search"
businessobject * * *
vault "Vehicle Project"
owner *;
```

Order Clause

The Order clause of the Add Cue statement defines the order in which the objects/relationships are displayed in relation to other cues: before, with, or after other cues. If more than one cue can apply to an object, eMatrix needs to know which cues to present. The before, with, and after ordering scheme establishes these priorities.

```
order -1|0|1
```

-1 is the lowest priority. These cues are applied first with any subsequent cue changes allowed.

0 ranks the cue in the order in which they are activated.

1 is the highest priority. Objects will change to these cues after any others are first applied.

Owner Clause

The Owner clause of the Add Cue statement assigns a cue for business objects that were created by a particular owner:

```
owner PATTERN
```

NAME_PATTERN defines the owner(s) for which you are creating a cue.

The Owner clause can use wildcard characters. You can also use multiple values to define the pattern. You can specify both the first and last name of an owner. For example, the following owner clause specifies all objects whose owner names begin with pat or thur:

```
add cue "Object Owner" owner pat*,thur*;
```

This statement might provide a cue for objects created by owners patricia, patty, and thurston.

Where Clause

The Where clause of the Add Cue statement is the most powerful and the most complicated. It searches for selectable business object properties. This involves examining each object for the presence of the property and checking to see if it meets the search criteria. If it does, the object is displayed with the cues applied; otherwise, it is ignored.

While the Where clause can test for equality, it can test for many other characteristics as well. The syntax for the Where clause details the different ways that you can apply the cue, using familiar forms of expression:

```
where "QUERY_EXPR"  
Or:  
where 'QUERY_EXPR'
```

QUERY_EXPR is the criteria to be used.

For detailed information, see [Where Clause](#) in Chapter 38.

Hidden Clause

You can mark the new cue as “hidden” so that it does not appear in the chooser in eMatrix. Users who are aware of the hidden cue’s existence can enter the name manually where appropriate. Hidden objects are accessible through MQL.

Property Clause

Integrators can assign ad hoc attributes, called Properties, to the cue. Properties allow associations to exist between administrative definitions that aren’t already associated. The property information can include a name, an arbitrary string value, and a reference to another administration object. The property name is always required. The value string and object reference are both optional. The property name can be reused for different object references, that is, the name joined with the object reference must be unique for any object that has properties.

```
add cue NAME property NAME [to ADMINTYPE NAME] [value STRING];
```

For additional information on properties, see *Overview of Administration Properties* in Chapter 24.

Copying or Modifying a Cue

You can modify any cue that you own, and copy any cue to your own workspace that exists in any user definition to which you belong. As an alternative to copying definitions, you can use the `set workspace` statement to make your workspace look like that of another user. Business Administrators can change their workspace to that of another user to work with cues that they do not own. See [Setting the Workspace](#) in Chapter 44 for details.

Copying (Cloning) a Cue Definition

After a cue is defined, you can clone the definition with the Copy Cue statement. Cloning a cue definition requires Business Administrator privileges, except that you can copy a cue definition to your own context from a group, role or association in which you are defined.

This statement lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy cue SRC_NAME DST_NAME [fromuser USER] [touser USER]  
[overwrite] [MOD_ITEM {MOD_ITEM}];
```

`SRC_NAME` is the name of the cue definition (source) to be copied.

`DST_NAME` is the name of the new definition (destination).

`USER` can be the name of a person, group, role, or association.

`overwrite` replaces an existing visual in the destination user.

The order of the `fromuser`, `touser` and `overwrite` clauses is irrelevant, but `MOD_ITEMS`, if included, must come last.

`MOD_ITEMS` are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

Modifying a Cue

The List Cue statement is used to display a list of all cues that are currently defined. It is useful in confirming the existence or exact name of a cue that you want to modify, since eMatrix is case-sensitive.

```
list cue [user USER_NAME];
```

`USER_NAME` can be the name of any person, group, role, or association. If the `user` modifier is used, the system displays the requested information relating to workspace objects of the person, group, role, or association indicated.

Users can list their own workspace objects or those of any group, role, or association to which they belong. Business Administrators can list the workspace objects of any users.

Use the list of all the existing cues along with the `Print` statement to determine the search criteria you want to change. Use the Modify Cue statement to add or remove defining clauses and change the value of clause arguments:

```
modify cue NAME [ITEM {ITEM}];
```

`NAME` is the name of the cue you want to modify.

ITEM is the type of modification you want to make. With the Modify Cue statement, you can use these modification clauses to change a cue:

active	The active option is changed to specify that the object is active.
notactive	The active option is changed to specify that the object is not active.
appliesto businessobject relationship all	The appliesto option is changed to reflect what the cue now affects.
type TYPE_PATTERN	The type is changed to the named pattern.
name PATTERN	The name is changed to the named pattern.
revision REVISION_PATTERN	The revision is changed to the named pattern.
color COLOR	The color is changed to the new color.
font FONT	The font is changed to the new font.
highlight COLOR	The highlight is changed to the new color.
vault PATTERN	The vault is changed to the pattern specified.
linestyle solid bold dashed dotted	The linestyle is changed to the new linestyle specified.
order -1 0 1	The order is modified.
owner PATTERN	The owner is changed to the pattern specified.
where QUERY_EXPR	The query expression is modified.
hidden	The hidden option is changed to specify that the object is hidden.
nothidden	The hidden option is changed to specify that the object is not hidden.
property NAME [to ADMINTYPE NAME] [value STRING]	The named property is modified.
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.

When making modifications, you simply substitute new values for the old. As you can see, each modification clause is related to the clauses and arguments that define the server.

Although the Modify Cue statement allows you to use any combination of criteria, no other modifications can be made. To change the cue name or remove the cue entirely, you must use the Delete Cue statement (see [Deleting a Cue](#)) and/or create a new cue.

For example, assume you have a cue named “Product Comparisons” with the following definition:

```
cue "Product Comparisons"
    type FormulaA
    name Lace
    revision 3
    color blue
    highlight yellow
    vault "Perfume Formulas"
    owner channel, taylor;
```

To this cue, you want to add another owner for the criteria. To make the change, you would write a Modify Cue statement similar to the following:

```
modify cue "Product Comparisons"  
    owner channel, taylor, cody;
```

This alters the cue so that it now appears as:

```
cue "Product Comparisons"  
    type FormulaA  
    name Lace  
    revision 3  
    color blue  
    highlight yellow  
    vault "Perfume Formulas"  
    owner channel, taylor, cody
```

Deleting a Cue

If a cue is no longer needed, you can delete it using the Delete Cue statement:

```
delete cue NAME;
```

NAME is the name of the cue to be deleted.

When this statement is processed, eMatrix searches the local list of existing cues. If the name is found, that cue is deleted. If the name is not found, an error message results.

For example, assume you have a cue named “Overdue Invoices” that you no longer need. To delete this cue from your area, you would enter the following MQL statement:

```
delete cue "Overdue Invoices";
```

After this statement is processed, the cue is deleted and you receive the MQL prompt for the next statement.

When a cue is deleted, there is no effect on the business objects or on queries performed by eMatrix. Cues are local only to the user’s context and are not visible to other eMatrix users.

Working With Tips

Object Tips Defined	846
Creating a Tip.....	847
Active Clause.....	847
Applies To Clause.....	847
Type Clause.....	847
Name Clause	848
Revision Clause.....	848
Vault Clause	848
Owner Clause	849
Where Clause	849
Expression Clause.....	849
Hidden Clause	849
Property Clause	850
Copying and/or Modifying a Tip	851
Copying (Cloning) a Role Definition.....	851
Modifying a Tip	851
Deleting a Tip	853

Object Tips Defined

Object Tips are small pop-up windows that appear from the eMatrix Navigator application when you hold the cursor briefly over any object in a browser. They are similar to the *Tool Tips* that appear over toolbar buttons in many programs to tell you what the button is for. However, unlike tool tips, you can define the contents of the Object Tip window to display the information you need most often about the objects you use. You can also use tips instead of Type Name and Revision to identify objects in eMatrix Navigator.

For example, you could include any attribute or basic property for an object that you frequently need to know, such as the current owner, or the last date the object was changed. This gives you a quick way to check basic data about an object without having to select Basics or Attributes from the menu or toolbar.

You can save Object Tip definitions by name (as personal settings) to turn on or off as you need them. A single tip may become part of several Views, being activated in one, and available in another. Refer to *Using View Manager* in the *eMatrix Navigator Guide* for more information on Views.

From eMatrix Navigator browsers, tips that quickly display pertinent information about objects can be very useful. Each user can create her/his own tips from eMatrix Navigator (or MQL). However, if your organization wants all users to use a basic set of similar tips consistently, it may be easier to create them in MQL, then copy the code to each user's personal settings (by setting context).

Tips can be defined and managed from within MQL or from the Visuals Manager in eMatrix Navigator. Once the tips are defined, individual users can turn them on and off from the eMatrix browsers as they are needed.

In the eMatrix Navigator browsers, they display on the Tips tab within the Visuals Manager window and in the View menu.

It is important to note that tips are all Personal Settings that are available and activated only when context is set to the person who defined them.

Creating a Tip

To define a new tip from within MQL, use the Add Tip statement:

```
add tip NAME [ADD_ITEM {ADD_ITEM}];
```

NAME is the name of the tip you are defining. Tip names cannot include asterisks.

ADD_ITEM specifies the characteristics you are setting.

When assigning a name to the tip, you cannot have the same name for two tips. If you use the name again, an error message will result. However, several different users could use the same name for different tips. (Remember that tips are local to the context of individual users.)

After assigning a tip name, the next step is to specify the conditions (ADD_ITEM) that each object must meet in order to display when the tip is turned on (activated). The following are Add Tip clauses:

[! in notin not]active
appliesto businessobject relationship
type TYPE_PATTERN
name PATTERN
revision REVISION_PATTERN
vault PATTERN
owner PATTERN
where QUERY_EXPR
expression EXPRESSION
[! not]hidden
property NAME on ADMIN [to ADMIN] [value STRING]

Each clause and the arguments they use are discussed in the sections that follow.

Active Clause

The Active clause of the Add Tip statement is used to indicate that the tip is active or not active (!active). The default is active.

Applies To Clause

The Applies To clause of the Add Tip statement indicates that the tip applies to business objects or relationships.

Type Clause

The Type clause of the Add Tip statement assigns a tip for a particular type of business object or relationship.

```
type TYPE_PATTERN
```

`TYPE_PATTERN` defines the types for which you are assigning a tip.

The Type clause can include more than one type by using multiple values to define the pattern. The Type clause can also use wildcard characters.

When listing multiple values as part of a pattern, separate each value with a comma and no spaces, or enclose any multi-word type value within quotation marks. If you include spaces, eMatrix reads the value as the next part of the tip definition.

Name Clause

The Name clause of the Add Tip statement assigns the names of objects to include in the tip.

```
name PATTERN
```

`PATTERN` defines the name(s) of objects to include in your tip.

The Name clause can include more than one name by using multiple values to define the pattern. The Name clause can also use wildcard characters. For example, the following definition displays a tip to display all business objects with names that start with “Inter”, or include the letters IBM, or include the words “International Business Machines”:

```
add tip "IBM"  
name Inter*, *IBM*, "International Business Machines";
```

When listing multiple values as part of a pattern, separate each value with a comma and no spaces, or enclose any multi-word type value within quotation marks (for example, “International Business Machines”). If you include spaces, eMatrix reads the value as the next part of the tip definition, as described in the Type clause.

Revision Clause

The Revision clause of the Add Tip statement assigns a tip for a particular revision of business objects.

```
revision REVISION_PATTERN
```

`REVISION_PATTERN` defines the revision for which you are assigning a tip.

The Revision clause can include more than one revision value and wildcards as in the other clauses that use patterns. Creating a tip that shows the most current change to an object can be very useful. This gives you a quick way to identify only the objects that have recently changed.

When listing multiple values as part of a pattern, separate each value with a comma and no spaces, or enclose any multi-word type value within quotation marks. If you include spaces, eMatrix reads the value as the next part of the tip definition.

Vault Clause

The Vault clause of the Add Tip statement assigns a tip for business objects that are in a particular or similar vault:

```
vault PATTERN
```

`PATTERN` defines the vault(s) for which you are assigning a tip.

The Vault clause can use wildcard characters. For example, the following definition displays a tip for all business objects that reside in the “Vehicle Project” vault:

```
add tip "Vault Search"  
businessobject * * *  
vault "Vehicle Project"  
owner *;
```

Owner Clause

The Owner clause of the Add Tip statement assigns a tip for business objects that were created by a particular owner:

```
owner PATTERN
```

NAME_PATTERN defines the owner(s) for which you are creating a tip.

The Owner clause can use wildcard characters. You can also use multiple values to define the pattern. You can specify both the first and last name of an owner. For example, the following owner clause specifies all objects whose owner names begin with pat or thur:

```
add tip "Object Owner" owner pat*,thur*;
```

This statement might provide a tip for objects created by owners patricia, patty, and thurston.

Where Clause

The Where clause of the Add Tip statement is the most powerful and the most complicated. It searches for selectable business object properties. This involves examining each object for the presence of the property and checking to see if it meets the search criteria. If it does, the object is displayed with the tips applied; otherwise, it is ignored.

While the Where clause can test for equality, it can test for many other characteristics as well. The syntax for the Where clause details the different ways that you can apply the tip, using familiar forms of expression:

```
where "QUERY_EXPR"
```

Or:

```
where 'QUERY_EXPR'
```

QUERY_EXPR is the criteria to be used.

For detailed information, see [Where Clause](#) in Chapter 38.

Expression Clause

The Expression clause of the Add Tip statement can be used to obtain or use information related to a tip, including business object data and also administrative object information.

EXPRESSION can be any select expression available for business objects. The object tip will display up to 100 characters. For detailed information, see the *Select Expressions* appendix in the *eMatrix Navigator Guide*.

Hidden Clause

You can mark the new tip as “hidden” so that it does not appear in the chooser in eMatrix. Users who are aware of the hidden tip’s existence can enter the name manually where appropriate. Hidden objects are accessible through MQL.

Property Clause

Integrators can assign ad hoc attributes, called Properties, to the tip. Properties allow associations to exist between administrative definitions that aren't already associated. The property information can include a name, an arbitrary string value, and a reference to another administration object. The property name is always required. The value string and object reference are both optional. The property name can be reused for different object references, that is, the name joined with the object reference must be unique for any object that has properties.

```
add tip NAME property NAME [to ADMINTYPE NAME] [value STRING];
```

For additional information on properties, see *Overview of Administration Properties* in Chapter 24.

Copying and/or Modifying a Tip

You can modify any tip that you own, and copy any tip to your own workspace that exists in any user definition to which you belong. As an alternative to copying definitions, you can use the `set workspace` statement to make your workspace look like that of another user. Business Administrators can change their workspace to that of another user to work with tips that they do not own. See [Setting the Workspace](#) in Chapter 44 for details.

Copying (Cloning) a Role Definition

After a tip is defined, you can clone the definition with the Copy Tip statement. Cloning a tip definition requires Business Administrator privileges, except that you can copy a tip definition to your own context from a group, role or association in which you are defined.

This statement lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy tip SRC_NAME DST_NAME [fromuser USER] [touser  
USER] [overwrite] [MOD_ITEM {MOD_ITEM}];
```

`SRC_NAME` is the name of the tip definition (source) to be copied.

`DST_NAME` is the name of the new definition (destination).

`USER` can be the name of a person, group, role, or association.

`Overwrite` replaces an existing visual in the destination user.

The order of the `fromuser`, `touser` and `overwrite` clauses is irrelevant, but `MOD_ITEMS`, if included, must come last.

`MOD_ITEMS` are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

Modifying a Tip

The List Tip statement is used to display a list of all tips that are currently defined. It is useful in confirming the existence or exact name of a tip that you want to modify, since eMatrix is case-sensitive.

```
list tip [user USER_NAME];
```

`USER_NAME` can be the name of any person, group, role, or association.

If the `user` modifier is used, the system displays the requested information relating to workspace objects of the person, group, role, or association indicated.

Users can list their own workspace objects or those of any group, role, or association to which they belong. Business Administrators can list the workspace objects of any users.

Use the list of all the existing tips along with the `Print` statement to determine the search criteria you want to change. Use the Modify Tip statement to add or remove defining clauses and change the value of clause arguments:

```
modify tip NAME [ITEM {ITEM}];
```

`NAME` is the name of the tip you want to modify.

ITEM is the type of modification you want to make. With the Modify Tip statement, you can use these modification clauses to change a tip:

active	The active option is changed to specify that the object is active.
notactive	The active option is changed to specify that the object is not active.
appliesto businessobject relationship	The appliesto option is changed to reflect what the tip now affects.
type TYPE_PATTERN	The type is changed to the named pattern.
name PATTERN	The name is changed to the named pattern.
revision REVISION_PATTERN	The revision is changed to the named pattern.
vault PATTERN	The vault is changed to the pattern specified.
owner PATTERN	The owner is changed to the pattern specified.
where QUERY_EXPR	The query expression is modified.
expression EXPRESSION	The select expression is changed to the expression specified.
hidden	The hidden option is changed to specify that the object is hidden.
nothidden	The hidden option is changed to specify that the object is not hidden.
property NAME [to ADMINTYPE NAME] [value STRING]	The named property is modified.
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.

These clauses are essentially the same ones that are used to define an initial tip except that Add property and Remove property clauses are included. When making modifications, you simply substitute new values for the old.

Although the Modify Tip statement allows you to use any combination of criteria, no other modifications can be made except the ones listed. To change the tip name or remove the tip entirely, you must use the Delete Tip statement (see [Deleting a Tip](#)) and/or create a new tip.

Deleting a Tip

If a tip is no longer needed, you can delete it using the Delete tip statement:

```
delete tip NAME;
```

NAME is the name of the tip to be deleted.

When this statement is processed, eMatrix searches the local list of existing tips. If the name is found, that tip is deleted. If the name is not found, an error message results.

For example, assume you have a tip named “Current Docs” that you no longer need. To delete this tip from your area, you would enter the following MQL statement:

```
delete tip "Current Docs";
```

After this statement is processed, the tip is deleted and you receive the MQL prompt for the next statement.

When a tip is deleted, there is no effect on the business objects or on queries performed by eMatrix. Tips are local only to the user’s context and are not visible to other eMatrix users.

42

Working With Toolsets

Toolsets Defined	856
Creating Toolsets.....	857
Active Clause.....	857
Program Clause.....	857
Method Clause.....	857
Hidden Clause	857
Property Clause.....	857
Copying and/or Modifying a Toolset.....	859
Copying (Cloning) a Toolset Definition	859
Modifying a Toolset Definition.....	859
Deleting a Toolset.....	861

Toolsets Defined

Your Business Administrator creates programs to perform specific functions. Some programs can run automatically when certain trigger events occur, such as the promotion of an object to a new state. Other programs could be executed to automate and standardize a common task, such as creating a cost analysis for a project, or creating a report on a project. Business Administrators create three types of programs:

- *Programs*, which can be executed without first selecting an object. This type of program might perform a query and generate a report on the found objects.
- *Methods*, which are programs that are associated with particular object type. For example, a method on a Product object might create a User Guide object and connect it to the Product automatically.
- *Wizards*, which are programs with a user interface that ask a series of questions and then execute their code. Wizards are similar to many Windows installation programs, and can be used in eMatrix to simplify or standardize a complex process. Wizards may be either stand-alone, like other Programs, or require a business object on which to act, and so become a method. Wizards are really a special type of eMatrix program and can be used as a program or a method within eMatrix.

When creating toolsets, you can access the list of all available Programs, Methods, and Wizards, then add a toolbar with buttons to perform these functions. You may want to consult your eMatrix Business Administrator when configuring your toolsets to determine the optimal selection for your individual use of eMatrix.

You can save Toolset definitions by name (as personal settings) to turn on or off as you need them. A single toolset may become part of several Views, being activated in one, and available in another. Refer to *Using View Manager* in the *eMatrix Navigator Guide* for more information on Views.

From eMatrix Navigator browsers, toolsets that provide easy execution from toolbar buttons can be very useful. Each user can create her/his own toolsets from eMatrix Navigator (or MQL). However, if your organization wants all users to use a basic set of similar toolsets consistently, it may be easier to create them in MQL, then copy the code to each user's personal settings (by setting context).

Toolsets can be defined and managed from within MQL or from the Visuals Manager in eMatrix Navigator. Once the toolsets are defined, individual users can turn them on and off from the eMatrix browsers as they are needed.

In the eMatrix Navigator browsers, toolsets display on the Toolsets tab page within the Visuals Manager window and in the View menu.

It is important to note that toolsets are all Personal Settings that are available and activated only when context is set to the person who defined them.

Creating Toolsets

To define a new toolset from within MQL, use the Add Toolset statement:

```
add toolset NAME [ ADD_ITEM {ADD_ITEM} ];
```

NAME is the name of the toolset you are defining. The toolset name cannot include asterisks.

ADD_ITEM specifies the characteristics you are setting.

When assigning a name to the toolset, you cannot have the same name for two toolsets. If you use the name again, an error message will result. However, several different users could use the same name for different toolsets. (Remember that toolsets are local to the context of individual users.)

After assigning a toolset name, the next step is to specify the conditions (ADD_ITEM) that must be met in order to display the toolbar button when the toolset is turned on (activated). The following are Add Toolset clauses:

[! in notin not]active
program NAME {,NAME}
method
[!not]hidden
property NAME on ADMIN [to ADMIN] [value STRING]

Each clause and the arguments they use are discussed in the sections that follow.

Active Clause

The Active clause of the Add Tip statement is used to indicate that the tip is active or not active (!active). The default is active.

Program Clause

The Program clause allows you to define which programs and/or wizards should be added to the Toolset. You can include multiple programs/wizards.

Method Clause

The Method clause allows you to define the method to be added to the Toolset. Only one method can be included in a toolset.

Hidden Clause

You can mark the new toolset as “hidden” so that it does not appear in the chooser in eMatrix. Users who are aware of the hidden toolset’s existence can enter the name manually where appropriate. Hidden objects are accessible through MQL.

Property Clause

Integrators can assign ad hoc attributes, called Properties, to the toolset. Properties allow associations to exist between administrative definitions that aren’t already associated. The

property information can include a name, an arbitrary string value, and a reference to another administration object. The property name is always required. The value string and object reference are both optional. The property name can be reused for different object references, that is, the name joined with the object reference must be unique for any object that has properties.

```
add toolset NAME property NAME [to ADMINTYPE NAME] [value STRING];
```

For additional information on properties, see *Overview of Administration Properties* in Chapter 24.

Copying and/or Modifying a Toolset

You can modify any toolset that you own, and copy any toolset to your own workspace that exists in any user definition to which you belong. As an alternative to copying definitions, you can use the `set workspace` statement to make your workspace look like that of another user. Business Administrators can change their workspace to that of another user to work with toolsets that they do not own. See [Setting the Workspace](#) in Chapter 44 for details.

Copying (Cloning) a Toolset Definition

After a toolset is defined, you can clone the definition with the `Copy Toolset` statement. Cloning a toolset definition requires Business Administrator privileges, except that you can copy a toolset definition to your own context from a group, role or association in which you are defined.

This statement lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy toolset SRC_NAME DST_NAME [fromuser USER] [touser USER]  
[overwrite] [MOD_ITEM {MOD_ITEM}];
```

`SRC_NAME` is the name of the toolset definition (source) to be copied.

`DST_NAME` is the name of the new definition (destination).

`USER` can be the name of a person, group, role, or association.

`Overwrite` replaces an existing visual in the destination user.

The order of the `fromuser`, `touser` and `overwrite` clauses is irrelevant, but `MOD_ITEMS`, if included, must come last.

`MOD_ITEMS` are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

Modifying a Toolset Definition

The `List Toolset` statement is used to display a list of all toolsets that are currently defined. It is useful in confirming the existence or exact name of a toolset that you want to modify, since eMatrix is case-sensitive.

```
list toolset [user USER_NAME];
```

`USER_NAME` can be the name of any person, group, role, or association. If the `user` modifier is used, the system displays the requested information relating to workspace objects of the person, group, role, or association indicated.

Users can list their own workspace objects or those of any group, role, or association to which they belong. Business Administrators can list the workspace objects of any users.

Use the list of all the existing toolsets along with the `Print` statement to determine the search criteria you want to change. Use the `Modify Toolset` statement to add or remove defining clauses and change the value of clause arguments:

```
modify toolset NAME [MOD_ITEM {MOD_ITEM}];
```

NAME is the name of the toolset you want to modify.

ITEM is the type of modification you want to make. With the Modify Toolset statement, you can use these modification clauses to change a toolset:

active	The active option is changed to specify that the object is active.
notactive	The active option is changed to specify that the object is not active.
add program NAME { ,NAME }	The named program is added to the toolset.
add method NAME { ,NAME }	The named method is added to the toolset.
remove program NAME { ,NAME }	The named program is removed from the toolset.
remove method NAME { ,NAME }	The named method is removed from the toolset.
remove all	All methods and programs are removed from the toolset.
hidden	The hidden option is changed to specify that the object is hidden.
nothidden	The hidden option is changed to specify that the object is not hidden.
property NAME [to ADMINTYPE NAME] [value STRING]	The named property is modified.
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.

These clauses are essentially the same ones that are used to define an initial toolset except that Add property and Remove property clauses are included. When making modifications, you simply substitute new values for the old.

Although the Modify Toolset statement allows you to use any combination of criteria, no other modifications can be made except the ones listed. To change the toolset name or remove the toolset entirely, you must use the Delete toolset statement (see [Deleting a Toolset](#)) and/or create a new toolset.

Deleting a Toolset

If a toolset is no longer needed, you can delete it using the Delete toolset statement:

```
delete toolset NAME;
```

NAME is the name of the toolset to be deleted.

When this statement is processed, eMatrix searches the local list of existing toolsets. If the name is found, that toolset is deleted. If the name is not found, an error message results.

When a toolset is deleted, there is no effect on the business objects or on queries performed by eMatrix. Toolsets are local only to the user's context and are not visible to other eMatrix users.

Working With Tables

Tables Defined	864
Creating a Table	865
Active Clause.....	865
Units Clause	865
Description Clause.....	866
Icon Clause.....	867
Column Clause	867
Hidden Clause	869
Property Clause	869
Copying and/or Modifying a Table	870
Copying (Cloning) a Table Definition	870
Modifying a Table	870
Evaluating a Table	873
Using Collections	873
Deleting a Table	875
Printing a Table	876

Tables Defined

eMatrix tables can be defined to display multiple business objects and related information. Each row of the table represents one business object. Expressions are used to define the columns of data that are presented about the business objects in each row. When you define a table, you determine the number and contents of your table columns.

In eMatrix, there are two kinds of tables:

- A *User* table is a user-defined template of columns that can be used when objects are displayed in the eMatrix Navigator (Details browser mode only), or in MQL with the Print Table statement.

User tables are created in MQL, or in eMatrix Navigator's Visuals Manager and displayed when eMatrix Navigator is in details mode. In eMatrix Navigator, tables that users create are available from the Views/Tables menu. This enables viewing different information about the same object with a single mouse-click.

Tables can be added as part of the definition for a customized View. Each time that View is activated, the Table defined for it displays.

Users can save Table definitions by name (as personal settings) to turn on or off as needed. A single table may become part of several Views, being activated in one, and available in another. Refer to *Using View Manager* in the *eMatrix Navigator Guide* for more information on Views.

From eMatrix Navigator browsers, tables that present information in a familiar format can be very useful. Each user can create her/his own tables from eMatrix Navigator (or MQL). However, if your organization wants all users to use a basic set of similar tables consistently, it may be easier to create them in MQL, then copy the code to each user's personal settings (by setting context).

- A *System* table is an Administrator-defined template of columns that can be used in custom applications. These tables are available for system-wide use, and not associated with the session context. Each column has several parameters where you can define the contents of the column, link data (href and alt), user access, and other settings. For example, a user could click on a link called Parts to display a system table containing a list of parts. The other columns in the table could contain descriptions, lifecycle states, and owners.

System tables are created by business administrators that have Table administrative access, and are displayed when called within a custom application.

System table columns can be role-based, that is, only shown to particular users. For example, the Parts table might have a Disposition Codes column that is shown only when a person is logged in as a user defined as a Design Engineer. Or a user defined as a Buyer might be shown a column in a table that is not seen by a Supplier user. When no users are specified in the command and table definitions, they are globally available to all users.

When business objects are loaded into a table, eMatrix evaluates the table expressions for each object, and fills in the table cells accordingly. Expressions may also apply to relationships, but these columns are only filled in when the table is used in a Navigator window. You can sort business objects by their column contents by clicking on the column header.

Creating a Table

To define a table from within MQL use the Add Table statement:

```
add table NAME user USER_NAME [ADD_ITEM {ADD_ITEM}] ;
```

Or

```
add table NAME system [ADD_ITEM {ADD_ITEM}] ;
```

NAME is the name of the table you are defining. Table names cannot include asterisks.

USER_NAME refers to a person, group, role, or association.

system refers to a table that is available for system-wide use, and not associated with the session context.

ADD_ITEM is an Add Table clause which provides additional information about the table. The Add Table clauses are:

[! in notin not]active
units [picas] points inches
description STRING_VALUE
icon IMAGE_PATH
column [label STRING_VALUE] COLUMN_TYPE_DEF [COLUMN_DEF_ITEM]
[! not]hidden
property NAME on ADMIN [to ADMIN] [value STRING]

The column clause must be used for each required column of the table, to specify the COLUMN_TYPE_DEF. All other clauses and subclauses are optional.

Each clause and the arguments they use are discussed in the sections that follow.

Active Clause

The Active clause of the Add Table statement is used to indicate that the table is active or not active (!active). The default is active.

Units Clause

The Units clause of the Add Table statement specifies the units of page measurement. There are three possible values: picas, points, or inches.

```
units picas  
Or  
units points  
Or  
units inches
```

Without a unit of measurement, eMatrix cannot interpret the values of any given header, footer, margin, or field size. Because picas are the default unit of measurement, eMatrix will automatically assume a picas value if you do not use a Units clause.

Picas are the most common units of page measurement in the computer industry. Picas use a fixed size for all characters. Determining the size of a field value is easy when using picas as the measurement unit. Simply determine the maximum number of characters that will be used to contain the largest field value. Use that value as your field size. For example, if the largest field value will be a six digit number, you need a field size of six picas. This is not true when using points.

Points are standard units used in the graphics and printing industry. A point is equal to 1/72 of an inch or 72 points to the inch. Points are commonly associated with fonts whose print size and spacing varies from character to character. Unless you are accustomed to working with points, measuring with points can be confusing and complicated. For example, the character "I" may not occupy the same amount of space as the characters "E" or "O." To determine the maximum field size, you need to know the maximum number of characters that will be used and the maximum amount of space required to express the largest character. Multiply these two numbers to determine your field size value.

Inches are common English units of measurement. While you can use inches as your unit of measurement, be aware that field placement can be difficult to determine and specify. Each field is composed of character string values. How many inches does each character need or use? If the value is a four-digit number, how many inches wide must the field be to contain the value? How many of these fields can you fit across a table page? Considering the problems involved in answering these questions, you can see why picas are a favorite measuring unit.

Description Clause

The Description clause of the Add Table statement provides general information about the function of the table. Since there may be subtle differences between tables, you can use the description clause to point out the differences.

You can distinguish your table in your selection of a table name. This consists of a character string value to identify the table being created and to reference it later. It should have meaning to the purpose of the table. If possible, avoid cryptic names. For example, "Cost Table" is a valid name, but it does not inform you of what costs you are presenting in the table.

Since the table name is too short to be very descriptive, you can include a Description clause as part of the table definition. This enables you to associate a prompt, comment, or qualifying phrase with the table being defined.

For example, if you were defining a table named "Cost Table," you might write an Add Table statement with a Description clause similar to one of the following. The information in each table might differ considerably.

add table "Cost Table" description "Provides daily operating costs of the department";
add table "Cost Table" description "Provides manufacturing costs for Widget A";
add table "Cost Table" description "Provides monthly costs for supporting Widget B";

When specifying a value for the description, enter a string of any length. However, the longer the string, the more difficult it may be to use.

Icon Clause

Icons help users locate and recognize items. You can assign a special icon to the new table or use the default icon. The default icon is used when in view-by-icon mode. Any special icon you assign is used when in view-by-image mode. When assigning a unique icon, you must use a .gif image file.

.giffilenames should not include the @ sign, as that is used internally by eMatrix.

Column Clause

Column clauses of the Add Table statement define each column in the table. It is made up of several subclauses. Each Column clause must have a businessobject, relationship, or set subclause, but all other subclauses are optional:

```
column [label STRING_VALUE] COLUMN_TYPE_DEF [COLUMN_DEF_ITEM]
```

If the label keyword is not used, the column heading is the expression.

STRING_VALUE is the text that is to appear in the heading of the column.

COLUMN_TYPE_DEF can be any of the following:

```
businessobject QUERY_WHERE_EXPRESSION
```

```
relationship QUERY_WHERE_EXPRESSION
```

```
set QUERY_WHERE_EXPRESSION
```

QUERY_WHERE_EXPRESSION is an expression that is evaluated on each object in the table with the results placed in the Table cell. This expression is constructed according to the syntax described in [Query Overview](#) in Chapter 38.

The QUERY_WHERE_EXPRESSION is used with businessobject, relationship, or set keyword. When applied to relationships, the information presented in the column will only be available from an *indented* table in eMatrix, where relationships are apparent. Relationship information will be blank in table cells that represent business objects and vice versa. A *flat* or regular table shows only the properties of the business objects it contains as expressed in the column definition. While the same table definition may be used as both a flat and indented table, generally the usage determines how the table is defined.

COLUMN_DEF_ITEM is a Column subclause that provides additional information about the value to be printed. These subclauses define information such as the size and scale of

the columns, the order in which the columns should be placed on the page, and the links used within the columns.

Column Definition	Meaning
size WIDTH HEIGHT	The default size of a column is determined by its contents, and the font size that eMatrix uses for dialogs. The defaults are recommended; however, you can set column sizes using the other column subclauses. The width and height can be explicitly set using the size clause with width and height values respectively:
minsize MIN_WIDTH MIN_HEIGHT	The minimum width and/or height of the column, for example: column units picas minsize 20 12
scale PERCENTAGE_VALUE	Percentage of the entire table to be used for this column. For example, use scale 25 for a 4-column table of equal column width.
href HREF_VALUE	The link data to the JSP. The Href link is evaluated to bring up another page. Many table columns will not have an Href value at all. The Href string generally includes a fully qualified JSP filename and parameters, which can contain embedded macros and expressions for mapping to database schema. Refer to Using Macros and Expressions in Configurable Components in Chapter 28 for more details.
alt _ALT_VALUE	Alternate text displayed until any image associated with the column is displayed and also as "mouse over text."
range RANGE_HELP_HREF_VALUE	For use in Web tables only to specify the JSP that gets a range of values and populates the column with the selected value.
update UPDATE_URL_VALUE	The URL address for updating the column.
program SORT_PROGRAM_NAME	Defines a program for sorting the table columns.
order NUMBER	The order of the column within the table. For example, order 3 would place the column as the third in the table.
sorttype alpha numeric other none	Determines how the column is sorted.
add user [USER_NAME all]	For use in Web forms only to specify who will be allowed access to the column.
add setting NAME VALUE	For use in Web forms only. Settings are general name/value pairs that can be added to a column as necessary. They can be used by JSP code, but not by hrefs on the Link tab. Also refer to Using Macros and Expressions in Dynamic UI Components in the <i>eMatrix Business Modeler Guide</i> for more details.
remove user [USER_NAME all]	For use in Web forms only to specify who will not be allowed access to the column.
remove setting NAME VALUE	For use in Web forms only to remove settings.

Column Definition	Meaning
autoheight [true false]	
autowidth [true false]	Autoheight and autowidth are on by default. However, if you set the size with the size clause, you can then use the autoheight and autowidth clauses to change back to the default, by specifying true.
edit [true false]	Determines whether users can edit cells in the column.
hidden [true false]	Determines whether the column is hidden.

For example, each of the following are valid column clauses:

```
column businessobject attribute["Target Cost"] - attribute["Actual Cost"]
```

```
column relationship name
```

```
column relationship attribute[Quantity]
```

Notice that selectable items can be operated on as numerical expressions.

Hidden Clause

You can mark the new table as “hidden” so that it does not appear in the chooser in eMatrix, which simplifies the end-user interface. Users who are aware of the hidden table’s existence can enter the name manually where appropriate.

Property Clause

Integrators can assign ad hoc attributes, called Properties, to the table. Properties allow associations to exist between administrative definitions that aren’t already associated. The property information can include a name, an arbitrary string value, and a reference to another administration object. The property name is always required. The value string and object reference are both optional. The property name can be reused for different object references, that is, the name joined with the object reference must be unique for any object that has properties.

```
add table NAME [system] property NAME [to ADMINTYPE NAME] [value STRING];
```

For additional information on properties, see *Overview of Administration Properties* in Chapter 24.

Copying and/or Modifying a Table

You can modify any table that you own, and copy any table to your own workspace that exists in any user definition to which you belong. As an alternative to copying definitions, you can use the `set workspace` statement to make your workspace look like that of another user. Business Administrators can change their workspace to that of another user to work with tables that they do not own. See [Setting the Workspace](#) in Chapter 44 for details.

Copying (Cloning) a Table Definition

After a table is defined, you can clone the definition with the Copy Table statement. Cloning a table definition requires Business Administrator privileges, except that you can copy a table definition to your own context from a group, role or association in which you are defined.

This statement lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy table SRC_NAME DST_NAME [fromuser USER] [touser USER]  
[overwrite] [MOD_ITEM {MOD_ITEM}];
```

SRC_NAME is the name of the table definition (source) to be copied.

DST_NAME is the name of the new definition (destination).

USER can be the name of a person, group, role, or association.

Overwrite replaces an existing visual in the destination user.

The order of the `fromuser`, `touser` and `overwrite` clauses is irrelevant, but `MOD_ITEMS`, if included, must come last.

MOD_ITEMS are modifications that you can make to the new definition. Refer to the table below for a complete list of possible modifications.

Modifying a Table

The List Table statement is used to display a list of all tables that are currently defined. It is useful in confirming the existence or exact name of a table that you want to modify, since eMatrix is case-sensitive.

```
list table [user USER_NAME | system] [select {FIELD_NAME {FIELD_NAME}}];
```

USER_NAME can be the name of any person, group, role, or association.

FIELD_NAME refers to selectable fields for tables. See the *Select Expressions* appendix in the *eMatrix Navigator Guide*.

If the `user` modifier is used, the system displays the requested information relating to workspace objects of the person, group, role, or association indicated.

Users can list their own workspace objects or those of any group, role, or association to which they belong. Business Administrators can list the workspace objects of any users.

Use the list of all the existing tables along with the Print statement to determine the search criteria you want to change. Use the Modify Table statement to add or remove defining clauses and change the value of clause arguments:

```
modify table NAME user USER_NAME [MOD_ITEM {MOD_ITEM}];
```

Or

```
modify table NAME system [MOD_ITEM {MOD_ITEM}];
```

NAME is the name of the table you want to modify.

USER_NAME refers to a person, group, role, or association.

system refers to a table that is available for system-wide use, and not associated with the session context.

MOD_ITEM is the type of modification you want to make. Each is specified in a Modify Table clause, as listed in the following table. Note that you need specify only the fields to be modified.

Modify Table Clause	Specifies that...
name NEW_NAME	The current table name is changed to the new name entered.
description VALUE	The current description value, if any, is set to the value entered.
units [picas] points inches	The current units definition is changed to the new units specified.
icon FILENAME	The image is changed to the new image in the file specified.
column delete COLUMN_NUMBER	The column identified by the given column number is removed from the table. To obtain the column number for a specific column, use the Print table statement. When the table definition is listed, note the number assigned to the column to delete.
column delete name COLUMN_NAME	The column identified by the given column name is removed from the table.
column modify COLUMN_NUMBER [label STRING_VALUE] COLUMN_TYPE_DEF [COLUMN_DEF_ITEM]	The column identified by the given column number is modified. To obtain the column number for a specific column, use the Print table statement. When the table definition is listed, note the number assigned to the column to delete.
column modify name COLUMN_NAME [label STRING_VALUE] COLUMN_TYPE_DEF [COLUMN_DEF_ITEM]	The column identified by the given column name is modified.
column COLUMN_DEF	A new column can be defined according to the column definition clauses and placed at the end of the table.
hidden	The hidden option is changed to specify that the object is hidden.

Modify Table Clause	Specifies that...
nohidden	The hidden option is changed to specify that the object is not hidden.
property NAME [to ADMINTYPE NAME] [value STRING]	The named property is modified.
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.

Each modification clause is related to the arguments that define the table. To change the value of one of the defining clauses or add a new one, use the Modify clause that corresponds to the desired change.

When modifying a table, you can make the changes from a script or while working interactively with MQL.

- If you are working interactively, perform one or two changes at a time to avoid the possibility of one invalid clause invalidating the entire statement.
- If you are working from a script, group the changes together in a single Modify Table statement.

Evaluating a Table

Integrators can use the evaluate table statement to include information from eMatrix tables in other applications. Information is returned for a single business object or relationship at a time. The business object ID or relationship ID is required and can be obtained using the print statement.

```
evaluate table NAME [user USER_NAME | system] businessobject ID;
```

Or

```
evaluate table NAME [user USER_NAME | system] relationship ID;
```

NAME is the name of the table you want to evaluate.

USER_NAME refers to a person, group, role, or association.

system refers to a table that is available for system-wide use, and not associated with the session context.

When a table is evaluated, all fields contained within the table for the specified business object ID or relationship ID are listed on your screen. For example, the following statement looks for information from a table named "RPO:"

```
evaluate table RPO businessobject 68104.11481.34617.772;
```

This statement would produce results similar to the following:

```
sismuth 9/12/98 62750 TKA Netting Co. Pr.1
```

Using Collections

The evaluate table command also allows a BUS_OBJ_COLLECTION_SPEC (as defined in *Business Object Collection Clause* in Chapter 1) to be given as an alternative to using the businessobject or relationship keywords:

```
evaluate table NAME [user USER_NAME | system] BUS_OBJ_COLLECTION_SPEC;
```

When using a BUS_OBJECT_COLLECTION_SPEC in an evaluate table command, the table columns are evaluated as summary data across the entire collection, returning a single row of information. As such, the column definitions should contain the set keyword, and use set-oriented expressions (that is: count, average, maximum, minimum, sum, product, standarddeviation, correlation). Any column definitions that include single business object expressions are ignored.

Example:

```
add table settable
    column label Sum set 'sum ( attribute[Priority] ) '
    column label Average set 'average ( attribute[Priority] ) '
    column label P1Count set 'count ( attribute[Priority] == 1 ) ';
evaluate table 'settable' set Escalate;
18.0 1.5 6
# Add a column whose expression is a "single bus obj" expression:
mod table settable add column label Priority set 'attribute[Priority] ';
# Which yields exactly the same evaluation:
evaluate table 'MySetTable' set Escalate;
18.0 1.5 6
```

If you want to perform an evaluate table with single business object expressions against each member of a collection, you need to create your own loop.

```
# Create a list of busId's for the set
set lObject [mql print set escalate select id dump |]
set lBusId [split $lObject \n]
# loop through the list of busId's and run evaluate table for each
# NOTE: use the keyword 'bus' to indicate a bus obj, not a rel ID.
foreach sBusId $lBusId {
    set sRow [mql evaluate table 'MyBOTable' bus $sBusId]
    puts $sRow
}
```

Deleting a Table

If a table is no longer required, you can delete it using the Delete Table statements

```
delete table NAME [user USER_NAME | system];
```

NAME is the name of the table to be deleted.

USER_NAME refers to a person, group, role, or association.

system refers to a table that is available for system-wide use, and not associated with the session context.

When this statement is processed, eMatrix searches the list of defined tables. If the name is found, that table is deleted. If the name is not found, an error message is displayed.

For example, to delete the table named “Income Tax Table,” enter the following:

```
delete table "Income Tax Table";
```

After this statement is processed, the table is deleted and you receive an MQL prompt for another statement.

Printing a Table

Use the Print Table statement to print information about the attributes of a specific table, including the number and characteristics of each table column.

```
print table NAME [user USER_NAME | system][SELECT];
```

NAME is the name of the table to be printed.

USER_NAME refers to a person, group, role, or association.

system refers to a table that is available for system-wide use, and not associated with the session context.

SELECT specifies a subset of the list contents. For a list of all the selectable fields for tables, see the *Select Expressions* appendix in the *eMatrix Navigator Guide*.

When this statement is processed, eMatrix searches the list of defined tables. If the name is found, that table information is printed. If the name is not found, an error message is displayed. For example, to print details about the table named “DescNote,” enter the following:

```
print table "DescNote";
```

The following is sample output:

```
MQL<28>print table 'DescNote';

table          DescNoteRes
inactive

#100000 column
label      Description
businessobject description
size        11 2
minsize    0 0
autoheight true
autowidth   true
editable    true
hidden     false
sorttype   none
user       all

#100001 column
label      Notes
businessobject attribute[Notes]
size        10 2
minsize    0 0
autoheight true
autowidth   true
editable    true
hidden     false
sorttype   none
user       all

nothidden
created Wed Oct 31, 2001 2:57:09 PM EST
```

modified Wed Feb 20, 2002 2:47:56 PM EST

Since tables have additional uses in support of dynamic UI modeling, the MQL print command suppresses the output of data that is not used. For example, if you print a table that is defined as a system object used for Web applications, the following selects will not be printed:

`size, minsize, scale, font, minwidth, minheight, absolutex, absolutey, xlocation, ylocation, width, and height.`

Conversely, when printing non-Web tables, parameters used only for Web-based tables are suppressed from the output:

`href, alt, range, update, and settings`

Working With Views

Views Defined	880
Creating a View.....	881
Active Clause.....	881
Hidden Clause	881
Property Clause.....	881
Copying a View	883
Setting the Workspace.....	884
Modifying a View.....	885
Deleting a View	887

Views Defined

When we look at some objects, we always need to know certain information and prefer that we see it in a similar format. From the eMatrix Navigator, customized Views offer a flexible and convenient way to *package* frequently used sets of Visuals (filters, cues, tips, toolsets and tables). For example, for each new revision of a product component, you might want to:

- Identify the latest change made with a red arrow cue
- List who performs work on it and who owns it with a quick object tip
- Filter out all the old revisions
- Always use the BOM Details table in the Details browser
- Run a program to create a new revision on command (with a tool button)

In eMatrix Navigator you can use View Manager to make your filter, cue, tip, toolset and table selections *only once*, name the *View* and save it. The next time you needed to create a new product revision, you select the object, select the named View and do your work.

In eMatrix Navigator, these Views are listed in the View menu by the names you give them, for you to use over and over again on any object/s you select. In MQL, you can use the List View statement to see all the available views.

Views can be defined and managed from within MQL or from the Visuals Manager in eMatrix Navigator. You cannot create a named View until you have several filters, cues, tips, toolsets or tables available from which to choose. Once the views are defined, individual users can turn them on and off from MQL or eMatrix Navigator browsers as they are needed.

From eMatrix Navigator, views that customize or standardize a display can be very useful. Each user can create her/his own views from eMatrix Navigator (or MQL). However, if your organization wants all users to use a basic set of similar views consistently, it may be easier to create them in MQL, then copy the code to each user's personal settings (by setting context).

In the eMatrix Navigator, the list of named views displays in the View Manager or in the in the View menu.

It is important to note that views are all Personal Settings that are available and activated only when context is set to the person who defined them.

Creating a View

To define a new view from within MQL, use the Add View statement:

```
add view NAME [ADD_ITEM {ADD_ITEM}];
```

NAME is the name of the view you are defining. View names cannot include asterisks.

ADD_ITEM specifies the characteristics you are setting.

When assigning a name to the view, you cannot have the same name for two views. If you use the name again, an error message will result. However, several different users could use the same name for different views. (Remember that views are local to the context of individual users.)

After assigning a view name, the next step is to specify which visuals or other conditions should be included in the view (ADD_ITEM) when the view is turned on (activated). The following are Add View clauses:

filter NAME
cue NAME
tip NAME
toolset NAME
table NAME
[! in notin not]active MEMBER_TYPE NAME {,NAME}
[! not]hidden
property NAME on ADMIN [to ADMIN] [value STRING]

The filter, cue, tip, toolset and table clauses require only the accurate name(s) of each one to be entered.

Active Clause

The Active clause of the Add View statement is used to indicate which members of the view are active or not active (!active). The default is active. For example, if you want a filter named NoClosed to be inactive, you could use the following active clause:

```
notactive filter NoClosed
```

Hidden Clause

You can mark the new view as “hidden” so that it does not appear in the chooser in eMatrix. Users who are aware of the hidden view’s existence can enter the name manually where appropriate. Hidden objects are accessible through MQL.

Property Clause

Integrators can assign ad hoc attributes, called Properties, to the view. Properties allow associations to exist between administrative definitions that aren’t already associated. The property information can include a name, an arbitrary string value, and a reference to another administration object. The property name is always required. The value string and

object reference are both optional. The property name can be reused for different object references, that is, the name joined with the object reference must be unique for any object that has properties.

```
add view NAME property NAME [to ADMINTYPE NAME] [value STRING];
```

For additional information on properties, see *Overview of Administration Properties* in Chapter 24.

Copying a View

After a view is defined, you can clone the definition with the Copy View statement. Cloning a view definition requires Business Administrator privileges, except that you can copy a view definition to your own context from a group, role or association in which you are defined.

Copying a View from one user to another is more complicated than for other Visuals because Views have members that need to be copied as well.

This statement lets you duplicate defining clauses with the option to change the value of clause arguments:

```
copy view SRC_NAME DST_NAME [fromuser USER] [touser USER]
[overwrite] [changename VISUAL OLD_NAME NEW_NAME] [MOD_ITEM
{MOD_ITEM}];
```

SRC_NAME is the name of the view definition (source) to be copied.

DST_NAME is the name of the new definition (destination).

USER is the name of a person, group, role, or association.

Overwrite replaces an existing member in the destination user.

VISUAL can be any one of the following: filter, tip, cue, toolset, table.

OLD_NAME is the current name of the member visual.

NEW_NAME is the name that the member visual will have after the copy command is executed.

The order of the `fromuser`, `touser`, `overwrite`, and `changename` clauses is irrelevant, but `MOD_ITEMS`, if included, must come last.

MOD_ITEMS are modifications that you can make to the new definition. Refer to the table in [Modifying a View](#) for a complete list of possible modifications.

When a View is copied, all its members are copied also. If a member of the same name exists in the target user, the command aborts, unless overwrite or changename is specified. If the command aborts, the database is left unchanged.

To avoid naming conflicts, you can specify a change of name when a member is copied. This is done with the changename keyword. For example:

```
copy view View1 ViewTable fromuser purcell touser Engineer
changename filter Bugs PurcellBugs changename tip RelationName
PurcellRelationName;
```

where View1 is a View belonging to person purcell containing a filter named Bugs and a tip named RelationName. This command will create a View named ViewTable in role Engineer. The filter Bugs will be copied to a filter named PurcellBugs. The tip RelationName will be copied to a tip named PurcellRelationName.

All other members of the View, if any, will be copied to the Role with their names unchanged. If a View with that name already exists or if a Filter of that name exists, etc., the command will abort and the database will be left unchanged.

Setting the Workspace

The `set workspace` command allows you to change the Workspace currently active in MQL so that the Workspace of some other User (person, group, role or association) is visible.

Users can change to the Workspace of groups, roles, and associations to which they belong. Business Administrators can change to the Workspace of any group, role, association, or person.

The syntax for this command is:

```
set workspace user USERNAME
```

This command affects the behavior of all commands to which Workspace objects are applicable, including:

- all commands specific to Workspace objects (for example, add/modify/delete filter)
- `expand bus` (affects which filters are used to control output)

`USERNAME` is the name of a person, group, role or association.

When users (other than Business Administrators) set Workspace to that of a group, role, or association, they cannot use commands that modify the Workspace, that is, the `add`, `modify`, and `delete` commands for Workspace objects. This restriction enforces the rule that only Business Administrators are permitted to change the Workspace of a group, role or association.

Note that `set context user USERNAME` will change the context to that of `USERNAME` regardless of an earlier invocation of `set workspace`.

Modifying a View

You can modify any view that you own, and copy any view to your own workspace that exists in any user definition to which you belong. As an alternative to copying definitions, you can use the `set workspace` statement to make your workspace look like that of another user. Business Administrators can change their workspace to that of another user to work with views that they do not own. See [Setting the Workspace](#) for details.

The List View statement is used to display a list of all views that are currently defined. It is useful in confirming the existence or exact name of a view that you want to modify, since eMatrix is case-sensitive.

```
list view [user USER_NAME];
```

USER_NAME can be the name of any person, group, role, or association. If the `user` modifier is used, the system displays the requested information relating to workspace objects of the person, group, role, or association indicated.

Users can list their own workspace objects or those of any group, role, or association to which they belong. Business Administrators can list the workspace objects of any users.

Use the list of all the existing views along with the `Print` statement to determine the search criteria you want to change. Use the Modify View statement to add or remove defining clauses and change the value of clause arguments:

```
modify view NAME [MOD_ITEM {MOD_ITEM}];
```

NAME is the name of the view you want to modify.

MOD_ITEM is the type of modification to make. Each is specified in a Modify View clause, as listed in the following table. Note that you need to specify only fields to be modified.

Modify View Clause	Specifies that...
<code>add filter NAME { ,NAME}</code>	The named filter is added.
<code>remove filter NAME { ,NAME}</code>	The named filter is removed.
<code>add cue NAME { ,NAME}</code>	The named cue is added.
<code>remove cue NAME { ,NAME}</code>	The named cue is removed.
<code>add tip NAME { ,NAME}</code>	The named tip is added.
<code>remove tip NAME { ,NAME}</code>	The named tip is removed.
<code>add toolset NAME { ,NAME}</code>	The named toolset is added.
<code>remove toolset NAME { ,NAME}</code>	The named toolset is removed.
<code>add table NAME { ,NAME}</code>	The named table is added.
<code>remove table NAME { ,NAME}</code>	The named table is removed.
<code>active MEMBER_TYPE NAME { ,NAME}</code>	The named MEMBER_TYPE (filter, cue, tip, toolset, table) is made active.
<code>inactive MEMBER_TYPE NAME { ,NAME}</code>	The named MEMBER_TYPE (filter, cue, tip, toolset, table) is made inactive.
<code>hidden</code>	The hidden option is changed to specify that the object is hidden.

Modify View Clause	Specifies that...
nohidden	The hidden option is changed to specify that the object is not hidden.
property NAME [to ADMINTYPE NAME] [value STRING]	The named property is modified.
add property NAME [to ADMINTYPE NAME] [value STRING]	The named property is added.
remove property NAME [to ADMINTYPE NAME] [value STRING]	The named property is removed.

As you can see, each modification clause is related to the clauses and arguments that define the view.

Although the Modify View statement allows you to use any combination of criteria, no other modifications can be made except the ones listed. To change the view name or remove the view entirely, you must use the Delete View statement (see [Deleting a View](#)) and/or create a new view.

Deleting a View

If a view is no longer needed, you can delete it using the Delete View statement:

```
delete view NAME;
```

NAME is the name of the view to be deleted.

When this statement is processed, eMatrix searches the local list of existing views. If the name is found, that view is deleted. If the name is not found, an error message results.

When a view is deleted, there is no effect on the business objects or on queries performed by eMatrix. Views are local only to the user's context and are not visible to other eMatrix users.

A

Macros Appendix

Using Macros	890
Macro Categories.....	892
Intrinsic Macros.....	892
Business Object Identification Macros.....	893
Format Macros.....	895
Lifecycle Check and Action Macros.....	895
Wizard Macros.....	895
Program Range Macros.....	896
Dynamic UI Macros	897
Connection Macros.....	898
Workflow Macros	899
JPO Macros	899
Trigger Macros	901
Trigger Macro Categories.....	901
Trigger Event Macros	906
Attribute Event Macros	906
Business Object Event Macros.....	906
Connection Event Macros.....	909
State Event Macros	910
Process Event Macros.....	912
Activity Event Macros	913
Query Event Macros	914
Macro Processing and Command Syntax	915

Using Macros

Before eMatrix version 6, macros were the only way in which eMatrix could pass information as variables to the command line. In version 6, the *Runtime Programming Environment (RPE)* was added. (See [Using the Runtime Program Environment](#) in Chapter 19.) Since macros are also generated in the RPE, and because it provides greater flexibility, use of the RPE is strongly encouraged. Yet, macros continue to be supported and provide a relatively easy way to work with variables. For example, a program used as the edit program in a Format would still likely use a macro to get the name of the checked in file to be edited. The main limitation to macros, however, is that they are available only for the program that calls them. When working with nested programs, the RPE enables you to pass values between programs, using the `get env MACRO`.

Although the operating system determines the valid syntax of the commands, the typical syntax used is:

```
command ${ "MACRO 1" } ${ "MACRO 2" } ...
```

When using macros be sure to enclose them in quotes to allow spaces. (An exception to this rule is the [OBJECT](#) macro.)

Macros are a simple name/value string substitution mechanism, where the macro value is substituted for the macro name when used in a Program. This single one-pass string substitution process occurs just prior to program execution. These macros are also created as a variable of the same name in the RPE.

RPE variables can be redefined (and possibly unset) by subordinate programs, for example called programs, utility programs such as load and validate in wizards, trigger programs, range programs, etc. To minimize the potential for unwanted name collisions, you should develop a good naming convention for your RPE variables, and ALWAYS move macro variables into your RPE variables right away. For example, in method wizards, the first thing you should do in the opening frame's prologue program is to move TYPE, NAME, and REVISION into local wizard variables (typically something like `wizardname_TYPE`, `wizardname_NAME`, `wizardname_REV`).

Refer to the discussion of [Macro Processing and Command Syntax](#) for more information.

In eMatrix, Programs can be used in several ways:

- As a standalone program or Tool
- As a Method
- As a view, edit, or print program in a Format
- As a lifecycle check or action
- As a Trigger check, override, or action program
- As an attribute's program range
- As a Wizard load, validate, prologue, epilogue, or button program
- As a Workflow AutoActivity

Different macros are populated with values depending on how the program is used. The table below shows each possible use of a Program in eMatrix, and the category of macros available to it. Macro category tables are provided later in this Appendix.

If a program is used as:	the available macros are:
a standalone program or Tool (Includes Wizards)	Intrinsic Macros
a Method (Includes Wizards)	Intrinsic Macros
	Business Object Identification Macros
a view, edit, or print program in a Format	Intrinsic Macros
	Format Macros
a lifecycle check or action	Intrinsic Macros
	Lifecycle Check and Action Macros
a Trigger check, override, or action program	Intrinsic Macros
	Standard Macros
	Depends on Event, See Trigger Macros
an attribute's program range	Intrinsic Macros
	Program Range Macros
a Wizard load, validate, prologue, epilogue, or button program.	Intrinsic Macros
	Wizard Macros
As a Workflow AutoActivity.	Intrinsic Macros
	AutoActivity Event Macros

Macro Categories

Intrinsic Macros

The table below lists macros that are available to ALL programs regardless of how it is invoked:

Intrinsic Macros	
Macro	Meaning
USER	User of the current context. When used within a shell command, USER is the user ID from the operating system, which is not necessarily the same as the eMatrix session user.
VAULT	Vault of the current context
INVOCATION	Tells how the program was invoked. Can have the following values: method, program, format, check, action, override, range, load, validate, prologue, epilogue, button, autoactivity, observer.
ARGCOUNT	Number of program parameters
TRANSACTION	Where the program was executed in terms of transaction boundaries. Potential values are read, update or an empty string, which means that it was executed outside of transaction boundaries.
HOST	Name of host machine that launched the current program
APPLICATION	Name of the application that launched the current program. Possible values are MQL, matrix, or BOS.
LANGUAGE	Language as defined in the initialization file in the variable MX_LANGUAGE_PREFERENCE.
HOME	User's home directory
PATH	Path statement of the operating system
MATRIXHOME	Directory that contains the eMatrix connection file
MATRIXPATH	Path to the eMatrix executable file

Intrinsic Macros	
Macro	Meaning
SELECTEDOBJECTS	<p>List of currently selected objects to be passed to any program. This macro is populated whenever a program or method is invoked from the toolbar, tool menu (Web Navigator only), right-mouse menu or Methods dialog. The macro is also populated when certain dialogs are launched via the <code>appl</code> command (appl icons, details, indented, star and indentedtable), as long as a dismiss program is defined. (See <i>Application Command</i> in Chapter 19). The macro's value consists of a single string of space-delimited business object IDs for the objects that are selected at the time the program or method is invoked. (For a method, this macro is redundant—it is equivalent to the OBJECTID macro—but it is populated nevertheless for consistency.)</p> <p>The SELECTEDOBJECTS macro can be read into a Tcl string or list variable as follows:</p> <pre># this reads the macro as a single Tcl string set sObjs [mql get env SELECTEDOBJECTS] # this reads the macro into a Tcl list. set env lObjs [split [mql get env SELECTEDOBJECTS]]</pre> <p>When no objects are selected, the macro is created, but holds an empty string.</p>
WORKSPACEPATH	Directory that is used as the base directory for upload/download. For programs/wizards run through the server, it is evaluated by combining the ematrix.ini settings MX_BOS_ROOT and MX_BOS_WORKSPACE, and appending a unique temporary directory for the session that runs it to avoid collisions across sessions. For desktop client configurations, it is set to the matrix.ini setting TMPDIR. These settings guarantee that the client has access to the files on the server.

Business Object Identification Macros

Most programs that use a business object have access to the following::

Business Object Identification Macros	
Macro	Meaning
TYPE	Business object type
NAME	Business object name
REVISION	Business object revision
OBJECT	Full business object specification (Type, Name, Revision). NOTE: Do not use double quotes for this macro, since it evaluates to more than a single item and each item (Type, Name, and Rev) in the output will be enclosed in double quotes.
OBJECTID	The system generated numerical identification of the object

Similarly, when used in the context of a connection, programs can access the following macros to identify objects on the ends of the relationship:

FromBusinessObject Macros	
Macro	Meaning
FROMOBJECT	Object Identifiers of Object on “from” end of connection
FROMTYPE	
FROMNAME	
FROMREVISION	
FROMOBJECTID	

ToBusinessObject Macros	
Macro	Meaning
TOOBJECT	Object Identifiers of Object on “to” end of connection
TOTYPE	
TONAME	
TOREVISION	
TOOBJECTID	

Use of the OBJECTID macro (as well as TOOBJECTID and FROMOBJECTID) is preferable to the use of TYPE, NAME and REVISION, or OBJECT variables in MQL/Tcl programs for the following reasons:

- Use of TYPE, NAME and REVISION requires a search through all vaults to locate the object.
- The TYPE, NAME, and REVISION combination is not guaranteed to be unique in a loosely-coupled environment or when used with Foreign vaults.

If TYPE, NAME, and REVISION must be used, include the [in VAULT] qualifier on the command line to limit the search to a single vault.

Format Macros

Programs that are used for file access can use Format macros.

Format Macros	
Macro	Meaning
FILENAME	Name of the first file in the default format or the selected file. During a checkout operation, includes the full target directory/filename. (For desktop applications, this is the final destination for the file. For ADK programs, this is the workspace directory on the server, prior to copying to the client.)
FILENAMES	Names of all the files in the default format, space delimited
FILENAME_ORIGINAL	Name of the first file in the default format or the selected file, as it is recorded in the database (hash name if using a hashed captured store), and also as it will be named on the client system after an ADK checkout operation.
FORMAT	Default format of the business object or the currently selected format
PROG	PROG is the full command line (with or without arguments) returned by Windows file associations.

Lifecycle Check and Action Macros

Programs used for lifecycle checks and actions can use the macros shown below.

Lifecycle Check and Action Macros	
Macro	Meaning
EVENT	Most recent transaction performed. For lifecycle checks or actions, EVENT can be <code>create</code> , <code>revisioned</code> , <code>copy</code> , <code>promote</code> , <code>demote</code> , or <code>change policy</code> .
OBJECT	Full specification of object (TYPE, NAME, and REVISION)
TYPE	Object's type
NAME	Object's name
REVISION	Object's revision

Wizard Macros

Wizards can use the following macros.

Wizard Macros	
Macro	Meaning
WIZARDNAME	Name of the Business Wizard.
FRAMENAME	Name of the current frame.
FRAMENUMBER	Number of the current frame in the frame sequence (excluding master and status frames). Frame numbers begin with 1. The status frame returns a value of 0.
FRAMETOTAL	Total number of frames in the frame sequence (excluding master and status frames).

Wizard Macros													
Macro	Meaning												
FRAME MOTION	<p>Current state of wizard processing:</p> <table> <tr> <td>start</td><td>The wizard has been started; the user has not yet clicked a button.</td></tr> <tr> <td>back</td><td>The user clicked the Back button.</td></tr> <tr> <td>next</td><td>The user clicked the Next button.</td></tr> <tr> <td>finish</td><td>The user clicked the Finish button.</td></tr> <tr> <td>repeat</td><td>The current frame has been redisplayed.</td></tr> <tr> <td>status</td><td>The current frame is the status frame of the wizard.</td></tr> </table>	start	The wizard has been started; the user has not yet clicked a button.	back	The user clicked the Back button.	next	The user clicked the Next button.	finish	The user clicked the Finish button.	repeat	The current frame has been redisplayed.	status	The current frame is the status frame of the wizard.
start	The wizard has been started; the user has not yet clicked a button.												
back	The user clicked the Back button.												
next	The user clicked the Next button.												
finish	The user clicked the Finish button.												
repeat	The current frame has been redisplayed.												
status	The current frame is the status frame of the wizard.												
WIZARDARG	<p>A single RPE variable that is created from the ARG string in an <code>appl wizard</code> command, which can be read by any of the wizard's supporting code to receive input from its 'caller.'</p> <p>If you want to pass multiple pieces of information from the caller to the wizard, you would need to format the information into a single string with an identifiable delimiter (for example, ' ').</p> <p>For example, to pass a businessobject type, current state and owner, you could collect them into a Tcl variable <code>sArgString</code> as follows:</p> <pre>set sArgString "\$sType \$sState \$sOwner" appl wizard bus \$sBusId MyWizard \$sArgString</pre> <p>Then, the wizard program that reads the WIZARDARG RPE variable could split it up again as follows:</p> <pre>set sArgString [mql get env WIZARDARG] set lArgList [split \$sArgString] set sType [lindex \$lArgList 0] set sState [lindex \$lArgList 1] set sOwner [lindex \$lArgList 2]</pre>												

Program Range Macros

Attribute range programs can use the following macros.

Program Range Macros	
Macro	Meaning
OBJECT	Owning Information for Objects. Note that OBJECTID is NOT available.
TYPE	
NAME	
REVISION	
RELID	Owning Information for Relationships

Program Range Macros	
Macro	Meaning
EVENT	EVENT will be “attribute choices” or “attribute check”. When EVENT equals “attribute check”, the ATTRVALUE macro will also be provided.
ATTRVALUE	The value of the attribute as set for the specified business Object. This macro is only provided when EVENT equals “attribute check”
ATTRNAME	The attribute name.
ATTRTYPE	The type of attribute. Value can be “String,” “Real,” “Integer”, “Date,” “Boolean.”

Owning Information Macros

The *Owning Information Macros* are available to programs used for attribute Ranges to provide the identity of the object or relationship that is using the attribute. When owner information is not supplied, the macros will not be provided to the program. This occurs if the program is used for something besides an attribute range, or if the MQL check attribute command is executed without the business object or connection ID clause.

Dynamic UI Macros

The following tables provide lists of macros used in the configuration parameters of the administrative menu and command objects found in the AEF. These menu and command objects are used for configuring the Menus/Trees/Actionbars in the Value Chain Portfolio applications that use them. These are the only macros currently supported for use in any dynamic UI component.

Directory Macros

The following table provides the list of directory specific macros used in the configuration setting.

Directory Macros	
Macro Name	Description
COMMON_DIR	To substitute the “common” directory below “ematrix” directory. The substitution is done with reference to any application-specific directory and it is relative to the current directory.
ROOT_DIR	To substitute the “ematrix” directory. The substitution is done with reference to any application-specific directory below “ematrix” and it is relative to the current directory.
SUITE_DIR	The macro to substitute the application-specific directory below “ematrix” directory. The substitution is done based on the “Suite” to which the command belongs and it is relative to the current directory.

Object Property Macros

The following table provides the list of Object specific macros supported by the dynamic UI components.

Object Property Macros	
Macro	Meaning
\${TYPE}	This will substitute the type name of the current business object. This Macro is currently used only in configuration of the Label for Root tree node. The tree type must be associated with a business object and require a valid business object id to process the macro substitution.
\${NAME}	This will substitute the name of the current business object. This Macro is currently used only in configuration of the Label for Root tree node. The tree type must be associated with a business object and require a valid business object id to process the macro substitution.
\${REVISION}	This will substitute the type name of the current business object. This Macro is currently used only in configuration of the Label for Root tree node. The tree type must be associated with a business object and require a valid business object id to process the macro substitution.

Connection Macros

Program objects that are executed as methods have two macros available when launched from a Navigator window.

Connection Macros	
Macro	Meaning
CONNECTIONID	The relationship ID of the connection next to the selected child business object in the Navigator browser.
CONNECTIONDIRECTION	The relationship direction (<code>to</code> or <code>from</code>) of the connection next to the selected child business object in the Navigator browser.

The connection macros are available only when the program is launched as a method from an object displayed in a Navigator window AND if the object is not the expanded object. Both macros will be empty if:

- no connection is displayed, as in the case when the selected business object is in a flat browser;
- more than one connection is shown, as when the selected object is in the center of a star browser or at the top of an indented browser.

In some cases the macros will not even be placed in the RPE. This situation occurs when a method is invoked programmatically (either from a program object or interactive MQL) or if a program is not executed as a method (i.e. from the toolbar).

Workflow Macros

Workflows can contain 4 types of programs: check, override, and action event triggers, as well as Autoactivity programs. The Macros available for Workflow trigger programs are listed in the [Process Event Macros](#) table. The macros below are available to Autoactivity programs.

Auto Activity Macros	
Macro	Meaning
PROCESS	These are Process Macros .
WORKFLOW	
WORKFLOWOWNER	
WORKFLOWID	
AUTOACTIVITYTYPE	Name of automated activity definition defined in Business Modeler
AUTOASSIGNEE	Name of automated activity assignee
AUTOPROGRAMNAME	Name of automated activity program
INPUTARGS	Input arguments for the program object
AUTOACTIVITY	Name of automated activity instance
AUTOACTIVITYID	System generated activity ID
AUTOACTIVITYOWNER	Name of activity owner

JPO Macros

Java Program Objects have two macros available:

JPO Macros	
Macro	Meaning
CLASSNAME	Refers to the JPO name in the Java class definition.
CLASS:jpo_name	Refers to other JPO classes inside the code of a JPO, for example when extending a base class or newing up an object. jpo_name is the name of the JPO holding the class of interest

There are two problems with using schema object names for the JPO class. The first problem is that eMatrix schema object names are allowed to contain spaces and other non-alpha characters that would cause errors in the Java compiler. The second problem is dynamic changes to the schema object names would not be automatically made to the code within JPOs.

Special macros are used to map the schema object names to legal Java-compliant names (a process known as *name mangling*). These macros are also appropriately modified automatically when a schema object name is changed.

Use of these parameterized class names allows both dynamic loading of classes after the source has been changed without restarting the system and automatic generation of dependencies between multiple Java Program Objects.

For example, if there is JPO named “RADProject” and its corresponding Business Type RADProject derives from Project, then the Java code defining the class for RADProject would look something like:

```
public class ${CLASSNAME} extends ${CLASS:Project}{:}
```

During compilation, the macro \${CLASSNAME} is mangled into a name that includes the JPO name “RADProject” (plus a suffix of “_mxJPO” followed by a hashed encoding). The macro \${CLASS:Project} is mangled into the exact same name as it was for \${CLASSNAME} appearing in the Project JPO. In other words, the system does all the appropriate name mangling and the JPO programmer needs to be concerned only with the higher-level macros.

For example, the “Hello World” JPO ends up with a class name something like “Hello_World_mxJPOTwEAAAAFAAAA” (the hashed encoding at the end will vary).

Trigger Macros

Macro (and RPE variable) values remain the same between activation of the check program, the override program, and the action program. This means one needs to be careful when interpreting the meaning of macros. For example, there are macros called OWNER and NEWOWNER for the changeowner event, which makes clear sense for the check program and override program, but by the time the action program executes, the owner may have changed. No additional macro called OLDDOWNER is necessary, but the action program must use OWNER to mean “old owner” and NEWOWNER to mean “current owner.”

Trigger programs have access to *Intrinsic Macros*, as well as others that are applicable to the specific event. The tables that follow first categorize the macros and then list them for each Event that supports triggers.

Trigger Macro Categories

The following tables list macros in categories. These categories are referenced later for each trigger supported event. Standard macros are available to programs used for all trigger events.

Standard Macros	
Macros	Meaning
TIMESTAMP	The current date/time
EVENT	Most recent transaction performed. EVENT for triggers will equal any of the <i>Supported Events</i> in Chapter 20.

Context/Access Macros	
Macro	Meaning
USER	Current user
ACCESSFLAG	Access flag (True if USER will be allowed to perform event)
CHECKACCESSFLAG	Check access flag (False when access is not checked, such as during import)

Business Object Macros	
Macro	Meaning
Business Object Identification Macros	Object Identifiers

Business Object Macros	
Macro	Meaning
OWNER	Object Settings and descriptions
DESCRIPTION	
CURRENTSTATE	
ENFORCEDLOCKING	
LOCKFLAG	
LOCKER	

State Macros	
Macro	Meaning
ISENABLED	Boolean State Flags
ISDISABLED	
ISOVERRIDDEN	
ISCURRENT	
ISREVISIONABLE	
ISVERSIONABLE	
HASSCHEDULEDATE	
HASACTUALDATE	
AUTOPROMOTE	

Signature Macros	
Macro	Meaning
SIGNATURE	Current Signature Name
ISSIGNED	Boolean Signature Flags
ISAPPROVED	
ISREJECTED	
ISIGNORED	
EXISTINGSIGNER	If signed, the signer's Name.
EXISTINGCOMMENT	Associated comment.

Relationship Macros	
Macro	Meaning
RELTYPE	Relationship Name
RELID	Relationship ID, available for all relationship trigger events.
FROMMEANING	From side meaning
FROMTYPES	From business Types allowed (excluding child Types)
FROMALLFLAG	Boolean flag - True if all business Types allowed
FROMREVACTION	Revision action of Object at From end. Value=Float, Replicate, or None.
FROMCARD	Cardinality of From connection. Value=One or N
TOMEANING	To side meaning
TOTYPES	To business Types allowed (excluding child Types)
TOALLFLAG	Boolean flag -True if all business Types allowed
TOREVACTION	Revision action of Object at To end. Value=Float, Replicate, or None.
TOCARD	Cardinality of To connection. Value=One or N

Attribute Macros	
Macro	Meaning
ATTRNAME	Name of attribute
ATTRVALUE	Current value
ATTRTYPE	Type (Integer, Real, String, Boolean, or Date)
ATTRDEFAULT	Default value
ATTRCHOICES	List of value choices

Process Macros	
Macro	Meaning
PROCESS	Name of the process definition defined in Business Modeler
WORKFLOW	Name of workflow defined in eMatrix Navigator
WORKFLOWOWNER	Name of workflow owner defined in eMatrix Navigator
WORKFLOWID	System generated workflow ID

Interactive Activity Macros	
Macro	Meaning
ACTIVITYTYPE	Name of activity definition defined in Business Modeler
ASSIGNEE	Name of user that process has assigned to the activity in Business Modeler
ACTIVITY	Name of activity instance created in eMatrix
ACTIVITYID	System generated activity ID
ACTIVITYOWNER	Name of activity owner

Auto Activity Macros	
Macro	Meaning
AUTOACTIVITYTYPE	Name of automated activity definition defined in Business Modeler
AUTOASSIGNEE	Name of automated activity assignee
AUTOPROGRAMNAME	Name of automated activity program
INPUTARGS	Input arguments for the program object
AUTOACTIVITY	Name of automated activity instance
AUTOACTIVITYID	System generated activity ID
AUTOACTIVITYOWNER	Name of activity owner

Query Variables/Macros	
Variable	Meaning
TYPE_PATTERN*	The value entered in the Type field of the Find or Find Like windows, or the TYPE_PATTERN specified in MQL/ADK programs.
NAME_PATTERN*	The value entered in the Name field of the Find or Find Like windows, or the NAME_PATTERN specified in MQL/ADK programs.
REVISION_PATTERN*	The value entered in the Revision field of the Find or Find Like windows, or the REVISION_PATTERN specified in MQL/ADK programs.
VAULT_PATTERN*	The value entered in the Vault field of the Find or Find Like windows, or the VAULT_PATTERN specified in MQL/ADK programs.
OWNER_PATTERN	The value entered in the Owner field of the Find or Find like windows, or the OWNER_PATTERN specified in MQL/ADK programs.
WHERE_PATTERN	The value entered in the Where field of the Find or Find like windows, or the WHERE_PATTERN specified in MQL/ADK programs.
EXPANDTYPE	Boolean value specified in the query.
LIMIT	Integer value specified in the query. May be set as environment variable. The setting in the query trigger will overwrite the setting in the environment.

When resetting these as RPE variables in a ValidateQuery program, you must make them global.

Trigger Event Macros

In addition to the macros listed for each event, trigger programs have access to *Intrinsic Macros* that are available to all programs, as well as the trigger *Standard Macros*.

Attribute Event Macros

Attributes can be assigned a Modify Trigger which has access to the following macros:

Attribute Event Macros		
Event	Macros Available	Meaning
modify	Attribute Macros	
	ATTRDESCRIPTION	Description of the attribute as defined by Business Administrator.
	USER	Logged in user of session context.

Modify event triggers on an attribute are executed when that particular Attribute of a business object or connection is modified. However, if the Type or Relationship has any *modifyattribute* event triggers defined they are also launched when this or *any* attribute of the connection or business object is modified.

The *modifyattribute* event has many more macros available to it. Refer to the [modifyattribute](#) event listing for more information.

Business Object Event Macros

The table below shows the specific macros available for the trigger events which are related to Object Types.

Business Object Event Macros		
Event	Macros Available	Meaning
changename	Business Object Macros	
	Context/Access Macros	
	NEWNAME	target name
	NEWREV	target revision
changeowner	Business Object Macros	
	Context/Access Macros	
	NEWOWNER	Target owner
changepolicy	Business Object Macros	
	Context/Access Macros	
	NEWPOLICY	Target policy

Business Object Event Macros		
Event	Macros Available	Meaning
changetype	Business Object Macros	
	Context/Access Macros	
	NEWTYPE	Target type
changevault	Business Object Macros	
	Context/Access Macros	
	NEWVAULT	target vault
checkin	Business Object Macros	
	Context/Access Macros	
	FORMAT	target format (if blank, default format used)
	FILENAME	target file- Note that FILENAMES is not available since for each file the user selects, a separate checkin event is performed.
	APPENDFLAG	append flag (True = append, False = overwrite)
checkout	Business Object Macros	
	Context/Access Macros	
	FORMAT	target format (if blank, default format used)
	FILENAME	target file (if blank checkout all files) Note that FILENAMES is not available since for each file the user selects, a separate checkout event is performed.
connect	Business Object Macros	
	Context/Access Macros	
	Relationship Macros	
	CONNECTION	end of relationship (From or To)
copy (clone)	Business Object Macros	
	Context/Access Macros	
	NEWNAME	
	NEWREV	
	NEWVAULT	Target vault (if blank, source vault used)

Business Object Event Macros		
Event	Macros Available	Meaning
create	Unlike all other Business Object events, the “create” event has no Business Object on which to operate.	
	USER	These are 2 of the 3 Context/Access Macros . ACCESSFLAG is not available.
	CHECKACCESSFLAG	
	OBJECT	New Object’s Identifiers.
	TYPE	
	NAME	
	REVISION	
	OBJECTID	This Macro is only available to the Action trigger program, after the object has actually been created.
	POLICY	Target Policy
	VAULT	Target Vault
delete	Business Object Macros	
	Context/Access Macros	
disconnect	Business Object Macros	
	Context/Access Macros	
	Relationship Macros	
	CONNECTION	end of relationship (From or To)
grant	Business Object Macros	
	Context/Access Macros	
	GRANTEE	A comma separated list of grantees to which the object is being granted.
	GRANTOR	Person granting the object, basically the current context.
	GRANTEEACCESS	List of accesses being granted to the grantees.
	GRANTEESENITURE	Boolean indicating whether or not the grantee can sign a signature for the grantor.
	<i>The user needs “Grant” access to be able to grant an object to another user.</i>	
revoke	Business Object Macros	
	Context/Access Macros	
	GRANTEE	The grantee that is being revoked.
	GRANTOR	The grantor that is being revoked. It is not required to be the current context.
	<i>The user needs “Revoke” access to be able to revoke access on an object.</i>	

Business Object Event Macros		
Event	Macros Available	Meaning
lock	Business Object Macros	
	Context/Access Macros	
modifyattribute	Business Object Macros	
	Context/Access Macros	
	Attribute Macros	
	NEWATTRVALUE	target attribute value
	CHECKRANGEFLAG	Boolean check range flag - FALSE when range checking is not performed such as during import.
modifydescription	Business Object Macros	
	Context/Access Macros	
	NEWDESCRIPTION	target description
removefile	Business Object Macros	
	Context/Access Macros	
	FORMAT	target format (if blank, default format used)
	FILENAME	target file (if blank remove all files) Note that FILENAMES is not available since for each file the user selects, a separate removefile event is performed.
revision	Business Object Macros	
	Context/Access Macros	
	NEWREV	target revision
unlock	Business Object Macros	
	Context/Access Macros	

Connection Event Macros

The table below shows the available macros for each supported connection event.

Connection Event Macros		
Event	Macros Available	Meaning
create event	Relationship Macros	
	Context/Access Macros	
	FromBusinessObject Macros	
	ToBusinessObject Macros	

Connection Event Macros		
Event	Macros Available	Meaning
delete event	Relationship Macros	
	Context/Access Macros	
	FromBusinessObject Macros	
	ToBusinessObject Macros	
freeze event	Relationship Macros	
	Context/Access Macros	
	FromBusinessObject Macros	
	ToBusinessObject Macros	
modifyattribute event	Relationship Macros	
	Context/Access Macros	
	Attribute Macros	
	NEWATTRVALUE	target attribute value
	CHECKRANGEFLAG	check range flag
	FromBusinessObject Macros	
	ToBusinessObject Macros	
thaw event	Relationship Macros	
	Context/Access Macros	
	FromBusinessObject Macros	
	ToBusinessObject Macros	

State Event Macros

The following table shows the macros that can be used in lifecycle event triggers.

State Event Macros		
Event	Macros Available	Meaning
approve event	Business Object Macros	
	Context/Access Macros	
	State Macros	
	Signature Macros	
	STATENAME	target state
	COMMENT	associated comment

State Event Macros		
Event	Macros Available	Meaning
demote event	Business Object Macros	
	Context/Access Macros	
	State Macros	
	STATENAME	source state
	NEXTSTATE	target state, if demotion succeeds
disable event	Business Object Macros	
	Context/Access Macros	
	State Macros	
	STATENAME	target state
enable event	Business Object Macros	
	Context/Access Macros	
	State Macros	
	STATENAME	target state
ignore event	Business Object Macros	
	Context/Access Macros	
	State Macros	
	Signature Macros	
	STATENAME	target state
	COMMENT	associated comments
override event	Business Object Macros	
	Context/Access Macros	
	State Macros	
	STATENAME	target state
promote event	Business Object Macros	
	Context/Access Macros	
	State Macros	
	STATENAME	source state
	NEXTSTATE	target state, if promotion succeeds

State Event Macros		
Event	Macros Available	Meaning
reject event	Business Object Macros	
	Context/Access Macros	
	State Macros	
	Signature Macros	
	STATENAME	target state
	COMMENT	associated comments
schedule event	Business Object Macros	
	Context/Access Macros	
	State Macros	
	STATENAME	target state
	SCHEDULED	scheduled date and time
unsign event	Business Object Macros	
	Context/Access Macros	
	State Macros	
	Signature Macros	
	STATENAME	target state

Process Event Macros

The following macros are available to Process trigger programs.

Process Event Macros		
Event	Macros Available	Meaning
FinishProcess	Process Macros	
ReassignProcess	Process Macros	
ResumeProcess	Process Macros	
StartProcess	Process Macros	
StopProcess	Process Macros	
SuspendProcess	Process Macros	

Activity Event Macros

Macros available to Interactive and Automatic Activity triggers are shown in the tables that follow.

Interactive Activity Event Macros		
Event	Macros Available	Meaning
ActivateActivity	Process Macros	
	Interactive Activity Macros	
CompleteActivity	Process Macros	
	Interactive Activity Macros	
OverrideActivity	Process Macros	
	Interactive Activity Macros	
ReassignActivity	Process Macros	
	Interactive Activity Macros	
ResumeActivity	Process Macros	
	Interactive Activity Macros	
SuspendActivity	Process Macros	
	Interactive Activity Macros	

AutoActivity Event Macros		
Event	Macros Available	Meaning
ActivateAutoActivity	Process Macros	
	Auto Activity Macros	
CompleteAutoActivity	Process Macros	
	Auto Activity Macros	
OverrideAutoActivity	Process Macros	
	Auto Activity Macros	
ReassignAutoActivity	Process Macros	
	Auto Activity Macros	
ResumeAutoActivity	Process Macros	
	Auto Activity Macros	
SuspendAutoActivity	Process Macros	
	Auto Activity Macros	

Query Event Macros

Query triggers are quite different than other triggers. Refer to [Validate Query Trigger](#) in Chapter 20 for more information.

Query Event Macros		
Event	Macros Available	Meaning
Query	Intrinsic Macros	
	Standard Macros	
	Query Variables/Macros	

If the ValidateQuery program is a JPO, no macros are available as input arguments. To work around this issue, the ValidateQuery program should be a Tcl program that calls the JPO to do the actual work.

Macro Processing and Command Syntax

“Macro processing” involves the replacement of macro names with their current values to create the precise code that is executed from a Program object or with the `shell` command. Macro values are not evaluated unless and until they are actually used. For example, a modify attribute trigger will not cause an attribute range program to be executed unless the macro `${ATTRCHOICES}` is explicitly used by the program.

Macros can affect the surrounding text if the required characters have additional uses in programming, such as “\$” or “\”. Because of this, in order to be able to include a literal “\${” in your text, you must turn off the macro processing substitution on a case-by-case basis by using the escape character “\”.

For example, if `USER=guest` then:

```
output "The value of \${USER} is ${USER}"
```

results in the output:

```
The value of ${USER} is guest
```

But one may also want the “\$” in the text to be macro processed. This can be accomplished by using a pair of backslashes. For example, if `MATRIXHOME=tools\matrix-9.0\` then:

```
output "The file can be found in \\${MATRIXHOME}"
```

results in the output:

```
The file can be found in \tools\matrix-9.0\
```

So, to restate, macro processing of a backslash ():

- followed by a “\$”, results in a literal \$
- followed by another backslash, results in a single backslash
- in all other cases is treated literally

The consequence of this means that any backslashes that need to appear in pairs (like the beginning of a UNC path) will require escaping twice by using four backslashes. For example, one might use:

```
shell "\\\\"ourserver\Matrix9\bin\winnt\mql.exe -v";
```

Note that macro processing is done in a single pass. Therefore, the values substituted in for macro names are not subject to macro processing. For example, if:

`MATRIXHOME=\matrix9`, then one could use:

```
shell "${MATRIXHOME}\bin\winnt\mql.exe -v";
```

Also note that the Tcl language also recognizes the backslash (among other characters) as an escape character. This means any Tcl code in a Program may require additional escaping. Refer to Tcl documentation for information on escaping in Tcl.

Index

A B C D E F G H I J K L M N
O P Q R S T U V W X Y Z

Symbols

`${NAME}` 643
 `${REVISION}` 643
 `${TYPE}` 643
:using as escape character 817

A

abort on error 45
abort transaction 74
aborting scripts 45
abstract types 325
access
 controlled by rules 375
 defining in policy 397
 delegating 714
 denying 399
 granting 399, 714
 owner 386
 public 386
 revoking 715
 summary of all 251
 to business objects 248
 to session information 90
 user 386
 where used 251
 which ones take precedence 248
 with filter expression 245
access log 90, 91, 93
 enabling 91
accessing MQL 43
action macros 895

action trigger
 described 522
 transaction boundaries 518
activity
 assigning to multiple users 689
 reassigning 558
add alias 612
add association
 definition clause 300
 description clause 299
 hidden clause 301
 icon clause 300
 introduced 299
 property clause 301
add attribute
 default clause 311
 description clause 311
 icon clause 311
 introduced 309
 multiline clause 317
 property clause 317
 rule clause 312
 trigger clause 317
 type clause 309
add businessobject
 specifying attributes 703
 description clause 698
 image clause 699
 introduced 698
 owner clause 701, 711
 policy clause 701
 revision clause 701
 state clause 702, 711
 vault clause 700
add command
 alt clause 636

code clause 637
command clause 637
description clause 635
file clause 637
href clause 636
icon clause 636
introduced 635
label clause 636
property clause 638
setting clause 637

add cue
active clause 838
apliesto clause 838
color clause 839
font clause 839
hidden clause 840
highlight clause 839
introduced 837
linestyle clause 839
order clause 839
owner clause 840
property clause 840
vault clause 839
where clause 840

add filter
active clause 827
apliesto clause 827
hidden clause 830
owner clause 829
property clause 830
where clause 829

add form
color clause 589
description clause 588
field clause 591
definition subclauses 592
type subclauses 592

footer clause 590
header clause 589
hidden clause 595
icon clause 589
introduced 587
margins clause 590
property clause 595
rule clause 589
size clause 591
type clause 591
units clause 587

add format
creator clause 366
description clause 365
edit clause 366

icon clause 367
introduced 365
print clause 366
property clause 369
suffix clause 367
type clause 366
version clause 368
view clause 366

add group
assign clause 293
assign role clause 293
child clause 292
description clause 292
hidden clause 294
icon clause 292
introduced 291
parent clause 292
property clause 295
site clause 294

add history 753

add inquiry
argument clause 659
code clause 658
description clause 657
file clause 659
format clause 658
icon clause 657
introduced 657
pattern clause 657
property clause 659

add location
description clause 150
hidden clause 154
host clause 150
icon clause 151
password clause 151
path clause 152
permission clause 152
property clause 154
url clause 153
user clause 153

add menu
alt clause 648
command clause 649
description clause 647
href clause 648
icon clause 647
introduced 647
label clause 648
menu clause 648
property clause 649
setting clause 649

```

add page
  description clause 618
  file clause 617, 618
  hidden clause 619
  icon clause 618
  mime clause 619
  property clause 619
add person 268
  access all 270
  access none 270
  address clause 271
  assign clause 271
  assign group clause 273
  assign role clause 272
  certificate clause 275
  clauses 269
  comment clause 274
  disable email clause 276
  disable password clause 278
  enable email clause 275
  enable iconmail clause 276
  fax clause 276
  fullname clause 276
  hidden clause 277
  icon clause 277
  no password clause 278
  passwordexpired clause 278
  phone clause 279
  property clause 279
  site clause 279
  type clauses 280
  vault clause 277
add policy
  defaultformat clause 391
  description clause 388
  format clause 390
  hidden clause 406
  icon clause 389
  introduced 388
  property clause 406
  sequence clause 391
  state
    access items 399
    action subclause 395
    check subclause 396
    checkouthistory subclause 405
    icon subclause 405
    notify subclause 396
    owner subclause 397
    promote subclause 405
    public subclause 397
    revision subclause 404
route subclause 397
signature subclause 401
trigger subclause 403
user subclause 397
version subclause 404
state clause 394
store clause 405
type clause 389
add process
  and clause 552
  attribute clause 544
  automated
    attribute subclause 549
    description subclause 549
    introduced 548
    program subclause 550
    trigger subclause 551
    user subclause 549
    xcoord subclause 550
    ycoord subclause 550
  autostart clause 553
  description clause 543
  finish clause 552
  hidden clause 553
  icon clause 553
  interactive
    attribute subclause 545
    description subclause 545
    duration subclause 546
    introduced 544
    priority subclause 546
    rate subclause 547
    trigger subclause 548
    user subclause 545
    wizard subclause 546
    worklist subclause 547
    xcoord subclause 547
    ycoord subclause 547
  introduced 543
or clause 552
property clause 553
start clause 552
stop clause 553
subprocess clause 551
trigger clause 554
add program
  code clause 458
  description clause 461
  downloadable clause 463
  execute clause 463
  external 461
  file clause 462

```

hidden clause 464
 icon clause 312, 462, 589
 introduced 458
 java 461
 mql 461
 needsbusinessobject 462
 property clause 465
 rule clause 462
 usesexternalinterface clause 463
add property 605
add query
 businessobject clause 790
 expandtype clause 791
 introduced 789
 owner clause 791
 vault clause 791
 where clause 792
add relationship
 attribute clause 341
 description clause 343
 from
 cardinality subclause 346
 clone subclause 351
 introduced 344
 meaning subclause 346
 propagate connection subclause 352
 propagate modify subclause 351
 revision subclause 348
 type subclause 345
 icon clause 343
 introduced 341
 property clause 353
 to
 cardinality subclause 346
 clone subclause 351
 introduced 344
 meaning subclause 346
 propagate connection subclause 352
 propagate modify subclause 351
 revision subclause 348
 type subclause 345
add report
 description clause 567
 displayrule clause 571
 field clause 571
 definition subclauses 572
 type subclauses 571
 footer clause 569
 header clause 568
 hidden clause 580
 icon clause 568
 introduced 566
 margins clause 570
 property clause 580
 size clause 568
 units clause 566
add resource
 description clause 627
 file clause 627
 hidden clause 628
 icon clause 628
 introduced 627
 mime clause 628
 property clause 628
add role
 assign clause 293
 child clause 292
 description clause 292
 hidden clause 294
 icon clause 292
 parent clause 292
 property clause 295
 site clause 294
add rule
 description clause 375
 icon clause 375
 introduced 375
 owner clause 377
 property clause 376
 public clause 377
 user clause 377
add server
 connect clause 172
 description clause 172
 hidden clause 175
 icon clause 172
 password clause 172
 property clause 175
 timezone clause 172
 user clause 172
add set
 bus_obj_collection_spec clause 768
 hidden clause 768
 introduced 767
 member clause 767
 property clause 768
add site
 description clause 158
 hidden clause 159
 icon clause 158
 member clause 159
 property clause 159
add statement 57
add store

description clause 128
filename hashed clause 129
hidden clause 135
host clause 131
icon clause 128
indexspace clause 135
introduced 127
locked 131
password clause 134
path clause 128, 131
permission clause 132
property clause 135
tablespace clause 135
type clause 129
unlocked 131
url clause 134
user clause 134

add table
active clause 865, 881
column clause 867
definition subclauses 867
description clause 866
hidden clause 869
icon clause 867
introduced 865
property clause 869
units clause 865

add tip
active clause 847
appliesto clause 847
expression clause 849
hidden clause 849
order clause 849
owner clause 849
property clause 850
where clause 849

add toolset
active clause 857
hidden clause 857
property clause 857

add type
abstract clause 329
attribute clause 188, 327
derived clause 328
description clause 187, 326
form clause 329
icon clause 187, 327
introduced 187, 326
method clause 329
property clause 188, 331
trigger clause 330

add vault

description clause 114
file clause 116
hidden clause 117
icon clause 115
indexspace clause 116
interface clause 116
introduced 114
map clause 117
property clause 117
server clause 116
tablespace clause 116

add view
active clause 885
hidden clause 881
property clause 881

add wizard
code clause 422
description clause 423
downloadable clause 424
execute clause 424
external clause 423
file clause 423

frame
color subclause 427
epilogue subclause 429
icon subclause 430
observer subclause 433
prologue subclause 428
size subclause 427
status subclause 429
units subclause 426
widget subclause 430

frame clause 425
hidden clause 425
icon clause 423
introduced 422
mql clause 423
needsbusinessobject clause 423
property clause 425

add workflow
attribute clause 683
description clause 681
image clause 681
introduced 681
owner clause 683
vault clause 683

ADK
session monitoring 105
setting history logging off 86

ADK interfaces for tracing 98
administration access mask 271
administration properties 604

administration vault 113
 administrative objects 58

- adding 57
- comparing 225
- copying 57
- deleting 62
- exporting 196
- importing 209
- modifying 58
- printing 61

 administrative requirements 171
 alias

- adding 612
- introduced 610
- modifying 612
- server connect string 170

 alternate text

- table column 868

 append clause

- file checkin 740

 application command

- checkin 496
- checkout 496
- forms 496
- help 497
- introduced 493
- modal clause 497
- notify 495
- program clause 493
- reassign set 496
- relationshipattributes 496
- relationshiphistory 496
- sendmail set 497
- wait clause 497

 application user 280
 approve businessobject 746
 association

- copying 302
- creating 290, 302
- defining 298, 299
- definition 300
- deleting 303
- description 299
- hidden 301
- icon 300
- modifying 302
- name 299
- property 301
- using AND 301
- using for notifications 298
- using for signatures 298
- using multiple operators 301

 using not equal 301
 using OR 301
 asynchronous replication 148
 attribute

- access 312
- assigning to object types 308
- assigning to relationships 308
- checking 319
- comparing value of 812
- copying 320
- default 311
- defined 308
- defining 309
- deleting 322
- description 311
- icon 311
- in business object definition 703
- modifying 320
- multiline 317
- multiple ranges 315
- name 309
- pattern comparison 314
- program range 315
- property 317
- range program 472
- relational operator 312
- rule 312
- sorting 710
- trigger 317
- types 309

 authenticating users 263
 autoheight

- form 593
- table column 869
- widget 434

 autowidth

- form 593
- table column 869
- widget 434

 average

- on a query 814
- on a set 781

B

- b command option 44
- backslash 470, 915
 - using as escape character 817
- backup strategy 110
- baseline
 - creating 225
 - use to compare schema 226

Boolean operators 799
bootstrap file 44
bos.ini file. *See* initialization file.
browser
 adding an object 494
 adding toolsets 857
 defining cues 837
 defining views 881
 update 495
building initial database 54
bulk loading 186
business administrator 281
business object
 access (locking) 744
 access precedence 248
 adding to revision sequence 720
 approve 746
 attributes 703
 checking files in and out 738
 checking in files 738
 checking out files 741
 collection spec 64
 connecting 724
 copies vs. revisions 718
 copying 711
 deleting 721
 demoting 750
 description 698
 disabling 748
 disconnecting 725
 dump 709
 enabling 748
 exporting 200
 grouping into set 766
 identification macros 893
 ignore 747
 image 699
 importing 216
 limit clause 736
 listing field names 706
 locking 744
 modifying 711
 modifying state 746
 name 695
 not analyzed in compare 226
 override 749
 owner 701, 711
 policy 701
 prerequisite objects 694
 print to file 710
 printing data without field names 709
 printing field names 708
 printing info about 705
 promoting 750
 protecting (locking) 744
 recordseparator clause 736
 reject 747
 relationship prerequisites 694
 retrieving image 700
 retrieving image files 700
 revising 718
 revision 701
 revision designator 696
 saving expansion data 732
 selecting fields to view 706
 sort attributes 710
 state 702, 711
 state scheduled date 702
 terse clause 736
 type 695
 unlocking 744, 745
 unsign signature 748
 vault 113, 700
 viewing definition 705
business wizard. *See* wizard.

C

-c command option 45
captured file 125
captured store
 and FTP 134
 defined 125
 enabling Secure FTP for 136
 replication 148
 synchronizing 162
captured stores
 hashing filenames 125
cardinality 346
certificate clause 275
chained programs 502
changename access 254
changeowner access 254
changepolicy access 254
changetype access
 described 254
 for relationship rule 377
changevault
 setting for the system 83
changevault access 254
check attribute 319
check trigger 522
checkin
 parameters 496

checkin access 251
checkin businessobject 738
checking in files 404, 738
 append 740
 override store 740
checking out files 738, 741
checkout
 access 251
checkout businessobject 741
checkout set 496
child
 program 502
clauses
 defined 49
 syntax 49
clear all 54
clear vault 55, 119
clearing databases 54
clone rule 351
cloning. *See* copying.
code for piped programs 478
color
 background 589
 foreground 589
 form fields 593
 frame 427
 visual cue 839
 widget 433
columns
 definitions in table 867
command
 adding 635
 alt 636
 code 637
 command 637
 copying 639
 defining 635
 deleting 641
 description 635
 file 637
 href 636
 icon 636
 label 636
 list 639
 modifying 639
 name 635
 property 638
 setting 637
command line
 editing 37
 interface
 options 44
 options for eMatrix 506
 wizard 447
 using control characters 37
comments
 entering 49
 syntax 49
 using 54
commit transaction 74
compare command
 examples 232
 introduced 226
 verbose mode 230
comparing schema
 creating baseline 225
 examples 232
 getting more details 230
 introduced 225
 objects not included 226
 reading the report 229
 with baseline 226
concatenating strings 794
connect businessobject 724
connect string 170
connecting business objects 724
connection
 change direction 727
 IDs 359
 replacing objects on the ends 727
 See also relationships.
connection macros 898
connectionid 496
const 812
constraint
 setting for the system 84
context
 default values 668
 defined 668
 disabled password 671
 no password 670
 password 669
 person 669
 person type 669
 restoring 672
 setting 669
 temporary change 506, 671
 threads share 500
 vault 669
 with password 669
continue keyword 53
controlling transactions 74
conventions xxv
copy 57

copy association 302
copy attribute 320
copy businessobject 711, 712
copy form 597
copy format 370
copy page 620, 621
copy person 284
copy policy 408
copy process 555
copy program 510
copy relationship 354
copy report 304, 582
copy resource 629
copy rule 379
copy server 176
copy set 770
copy type 189, 332
copy wizard 450
copying
 association definition 302
 attribute definition 320
 business object definition 711
 command definition 639
 filter definition 831
 form definition 597
 format definition 370
 inquiry definition 660
 items 57
 menu definition 650
 person definition 284
 policy definition 408
 process definition 555
 program definition 510
 relationship definition 354
 report definition 582
 rule definition 379
 set definition 770
 table definition 870
 tip definition 851
 toolset definition 859
 type definition 332
 view definition 883
 visual cue definition 841
 visuals 304
 wizard definition 450
core session monitoring 105
core statistics 107
correlation
 on a query 815
 on a set 782
count
 on a query 814

 on a set 780
create access 253
creator user 268
cue. *See* visual cue.

D

-d command option 45
data types 309
database
 aliases for administrative work 171
 building 54
 cleaning up indices 119
 clearing 54
 comparing schema 225
 initial 54
 modifying 56
 tablespace names 116
 validating 88
database administrator
 and tablespace names 135
date
 current date/time 794
 modified 60
debugging 101
decimal settings for the system 85
default icon 128
defining
 association 298, 299
 attribute 309
 command 635
 form 587
 format 365
 group 291
 inquiry 657
 location 149
 menu 647
 policy 388
 process 543
 program 458
 query 789
 relationship 341
 report 566
 role 291
 server 171
 set 767
 site 158
 store 127
 table 865
 type 187, 326
 vault 114
 workflow 681

definitions
 creating 55
 order for creating 55
defragmenting database file 140
delegating access 714
delete 62
 delete access 251
 delete association 303
 delete attribute 322
 delete businessobject 721
 delete command 641
 delete cue 844
 delete filter 833
 delete form 599
 delete format 372
 delete history 758
 delete inquiry 663
 delete location 157
 delete mail 678
 delete menu 653
 delete page 622
 delete person 287
 delete policy 415
 delete process 561,562
 delete program 513
 delete property 606
 delete query 823
 delete relationship 358
 delete report 584
 delete resource 631
 delete rule 382
 delete server 177
 delete set 783
 delete site 161
 delete store 144
 delete table 875
 delete tip 853
 delete toolset 861
 delete type 192,335
 delete vault 122
 delete view 887
 delete widget 454
 delete wizard 454
delete workflow 691
deleting
 association 303
 attributes 322
 business objects 721
 command 641
 cues 844
 files 721
 filters 833
form 599
format 372
group 295
history 758
IconMail message 678
inquiry 663
items 62
location 157
menu 653
pages 622
person 287
policy 415
process 561
program 513
query 823
relationship connections 360
relationships 358
report 584
resource objects 631
role 295
rows in Oracle table 120
rule 382
server 177
set 783
site 161
store 144
table 875
tips 853
toolset 861
type 192,335
vault 122
view 887
visual cues 844
visuals 305
widget 454
wizard 454
workflow 691
demote access 253
demote businessobject 750
diagnostic tools 95
dialog
 including in a wizard 448
 with appl command 493
dialog output 499
dialog scripting language 493
disable access 254
disable businessobject 748
disable server 177,180
disabling
 business object 748
 event triggers 540,763
 server 177

disconnect businessobject 725
 disconnect connection 360
 dismiss program 493
 distributed database 178

- in vault clause 63

 distribution 168, 178
 division operator 793
 double quotes

- using with single quotes 817

 download command 444
 downloadable

- programs
 - eMatrixApplet windows disabled 500
 - threads share context 500

 dump

- printing without field names 709

E

edit

- table column 869

 editing

- command line 37
- control characters 37
- widget 435

 eMatrix error

- downloadable 463, 500
- downloadable clause 424
- invalid date/time 811
- sessions command 90
- table does not exist 90
- validate 88
- vault definition 283
- view does not exist 90

 eMatrix Navigator. See Web version.
 ematrix.ini file. See initialization file.
 eMatrixApplet. See Web version.
 enable access 254
 enable businessobject 749
 enabling

- business object 748
- event triggers 540, 763

 encryption for passwords 266, 267
 epilogue for frame 429
 error

- abort 45
- creating process 562
- message, hiding 526
- redirecting 46

 error command 477
 error log 90
 error message

- downloadable 424, 463, 500
- invalid date/time 811
- ORA-00987 170

 error messages

- sessions command 90
- table does not exist 90
- validate 88
- vault definition 283
- view does not exist 90

 escape character 817

- and Tcl 819

 eval statements 443
 evaluate expression 64
 evaluate inquiry 662
 evaluate query

- into set 821
- introduced 821
- onto set 821
- over set 821

 evaluate report 581
 evaluate table 873
 evaluating

- inquiry 662
- query 821
- report 581

 reserved words and selectables 812
 table 873
 event triggers

- action 403, 518, 522, 548, 551, 554
- action program 528
- activity event macros 913
- and recursion 535
- attribute event macros 520
- auto activity events 521
- blocking events 526
- business object event macros 906
- check 403, 522, 548, 551, 554
- check programs 526
- create event 523
- designing 520
- disabling 540, 763
- enabling 540, 763
- interactive activity events 521
- lifecycle actions 525
- lifecycle checks 525
- lifecycle events supported 403, 554
- macro categories 901
- macros 526, 901
- multi-trigger events 522
- override
 - in policies 403
 - in processes 548, 551, 554

introduced 522
programs 527

process event macros 912

process events 521

programs 526

recursion 535

relationship 524

relationship event macros 909

replacing events 527

scenarios 517

state event macros 910

state events 521

transaction boundaries 518

transaction model 519

types 521

user interaction in 524

workflow events 521

event viewer 94

events

- logging 539

exception file

- compare command 229
- export command 199

exclude file

- compare command 228
- export command 198

exec command 488, 509

execute access

- described 252
- for program rules 376

execute workflow 550

exit command 500

expand businessobject 728

- dump clause 734
- from 729
- introduced 728
- limit clause 736
- output clause 734
- recordseparator clause 736
- recurse to clause 731
- relationship clause 730
- select clause 733
- select relationship 359
- set clause 732
- tcl clause 736
- terse clause 736
- to 729
- type clause 730

expand set

- dump clause 776
- from 772
- into|onto set clause 775

introduced 772

output clause 776

relationship clause 772, 774

tcl clause 776

terse clause 776

to 773

type clause 774

expanding

- business objects 728
- objects in sets 772

expandtype clause 791

explicit

- transactions 74
- type characteristics 325

export

- !icon clause 197
- !mail clause 197
- !sets clause 197
- admin clause 196, 227
- creating baseline 225
- exclude clause 198
- into form clause 198
- introduced 196
- onto form clause 198

export businessobject

- !file 201
- !history 201
- !relationship 201
- !state 201
- introduced 200

export files, extracting information from 222

export workflow 203

exporting

- administrative objects 196
- business objects 200
- excluding information 201
- workflow 203

expression

- filters 245

expression access filter

- on policy state 398
- on rules 377

expressions

- arithmetic 793
- Boolean 798
- comparative 792
- complex Boolean 800
- evaluating 64
- if-then-else 801
- matchlist 797
- on sets 780
- query 792

set relationships 772
 smatchlist 797
 string concatenation 794
 substring 801
 unexpected results 812
 using const for exact equal 812
 using dates 794
 extending transaction boundaries 74
 external authentication 263
 external programs 461
 extract 222
 extracting from export files 222

F

field names
 listing 706
 printing 708
 field. *See* form.
 fields
 definitions in form 592
 definitions in report 572
 on form 591
 report 571
 types in form 592
 types in report 571
 file
 assigning extension 367
 checkin access 251
 checking into business object 738
 checking out of business object 741
 checkout access 251
 deleting 721
 exception 229
 exclude 228
 exporting and importing 223
 hashed names 129
 in store 124
 log 228
 map 228
 migrating 223
 output of print statement 710
 retrieving business object images 700
 retrieving workflow images 682
 running 53
 sending trace info to 95
 transfer in program or wizard 444
 filenames
 hashing 125
 filter
 active 827
 applies to 827

copying 831
 deleting 833
 expression for policy 398
 expression for rule 377
 hidden 830
 modifying definition 831
 name 827
 owner 829
 property 830
 query expressions 830
 where clause 829
 filter expression 245
 find *See* query.
 float revision/clone rule 350
 fonts, specifying for Tk 41
 footer
 form 590
 report 569
 foreign vault 115
 form
 adding 587
 color 589
 copying 597
 defined 586
 defining 587
 deleting 599
 description 588
 designing for Web and desktop versions 591, 593
 display 496
 field color 593
 field definition 592
 field starting point 594
 field types 592
 fields 591
 font 593
 footer 590
 for HTML/JSP applications 587
 header 589
 hidden 595
 icon 589
 margins 590
 modifying 597
 name 587
 printing 600
 property 595
 rule 589
 size 591
 types associated with 591
 units 587
 format
 adding 365

copying 370
 creator 366
 defined 364
 defining 365
 deleting 372
 description 365
 edit 366
 for page objects 623
 for resource objects 632
 hidden 368
 icon 367
 MIME type 368
 modifying 370
 name 365
 print 366
 property 369
 suffix 367
 type 366
 version 368
 view 366
 format macros 895
 frame

- color subclause 427
- definition 418
- epilogue subclause 429
- icon subclause 430
- modifying 452
- observer subclause 433
- prologue subclause 428
- sequence 439
- size subclause 427
- skipping 428
- status subclause 429
- units subclause 426
- widget
 - autoheight subclause 434
 - autowidth subclause 434
 - color subclause 433
 - drawborder subclause 435
 - edit subclause 435
 - font subclause 434
 - load subclause 432
 - multiline subclause 435
 - name subclause 431
 - password subclause 435
 - scroll subclause 435
 - size subclause 434
 - start subclause 434
 - upload subclause 435
 - validate subclause 432
 - value subclause 431
- widget subclause 430

freeze access
 described 252
 for relationship rule 376
freeze connection 360
 freezing relationship connections 360
 fromconnect access
 described 254
 for relationship rule 376
 fromdisconnect access
 described 255
 for relationship rule 376
fromset selectable for business objects 805
 FTP

- and locations 151
- and stores 134
- in add location statement 132, 151

 full user, defining 280

G

grant access 253
 granting access 714
 group

- assigning roles 293
- assigning to a person 273
- defining 291
- deleting definition 295
- hidden 317
- hierarchy 290, 292
- lifecycle example 288
- modifying definition 295
- multiple parents 291
- name 291
- reassigning workflow activity 558

 guest user 268

H

hashed filenames 129
 hashing filenames

- captured stores can use 125

 header

- form 589
- report 568

 help

- accessing 51
- with item commands 62

 help command 51, 62
 hidden

- table column 869

 hierarchy

- group 290, 292

role 290
history
 adding custom 753
 deleting 758
 excluding 754
 printing 754
 select 754
 setting for the system 86
href
 command 636
 menu 648
 table column 868

icon
 assigning 63
 location 151
 store 128
 wizard 423
 wizard frame definition 430

icon clause 63

IconMail
 carbon copy 676
 defined 674
 deleting message 678
 message text 676
 recipient 675
 renumbering messages 678
 send set 497
 sending 675
 subject 676

ID
 connection 359
 object 697
 relationship 359

ignore businessobject 747

ImageIcon
 assigning to business object 699
 assigning to workflow 681
 retrieving 682, 700

implicit
 transactions 74
 type characteristics 325

import
 business objects 216
 eMatrix Exchange Format files 209
 excluding information 217
 order 209
 properties 214
 servers 213
 strategy 223

workflow 213, 221
workspace 213
XML files 209

import 209
 !icon clause 211
 admin clause 210
 exclude clause 212
 list clause 210
 use map clause 212

import businessobject 216
 !file 217
 !history 217
 !relationship 217
 !state 217
 exclude clause 219
 from vault clause 217
 list clause 217
 preserve clause 219
 to vault clause 217
 use map clause 219

import workflow 221

in vault clause
 object identifier 63
inactive person 281
inches 426, 566, 587, 865
index 186
 disabling 186
 selects 191
 validate 191

index vault
 validate 119

index vault 119

indexed fields 808

ingested
 file 129
 store 125

initialization file
 MX_ANNOTATION_TYPE 334
 MX_ATTACHMENT_TYPE 334
 MX_MQL_TRACE 476
 MX_REPORT_TYPE 334

input
 from file 45
 redirecting 45
 script 45

inquiry
 adding 657
 argument 659
 code 658
 copying 660
 defining 657
 deleting 663

description 657
file 659
format 658
icon 657
list 660
modifying 660
name 657
pattern 657
property 659
instances of relationships 359
interactive mode 36
internal object 802
intrinsic macros 892
inventory store 140

J

Java
tracing 95
java programs 461
JPO macros 899

K

-k command option 45
keyword
defined 50
in expressions 812
in queries 812
in statements 50
syntax 50

L

language
page objects 623
resource objects 632
language alias
adding 612
defined 610
modifying 612
properties 610
LCD 484
doesn't support external authentication using
LDAP 262, 263
in vault clause 63
LDAP authentication 263
cipher setting 266
encryption for passwords 267
lifecycle 384
lifecycle check macros 895
link data

for table 868
link transition conditions 559
list 58
list admintype 59
after date clause 61
dump clause 60
name pattern 59
output clause 61
record_sep clause 61
select clause 60
tcl clause 61
listing
administrative objects 58
administrative types 59
business object fields 706
command 639
cues 841
inquiry 660
items 58
menu 650
property 606
store contents 140
system settings 87
table 870
tips 851
load program widget 432
local vault 115
location
access privileges 152
defined 148
defining 149
delete 157
deleting 157
description 150
hidden 154
host 150
icon 151
password 151
path 152
permission 152
port 151
property 154
protocol 151
purge 157
url 153
user 153
lock access 252
lock businessobject 744
lockout, password 265
log file
compare command 228
export command 198

report for compare command 229
logging events 539
logging trigger events 539
long match 798
Loosely-Coupled Databases 484

M

macro
 for object name 643
 for object revision 643
 for object type 643
 for select expressions 643
macro processing 470, 915
macros
 \${} 440
 action 895
 activity events 913
 attribute events 520
 attribute range program 472
 available to all programs 892
 business object events 906
 business object identification 893
 categories 901
 connection 898
 event triggers 526, 901
 for programs 421
 format 895
 intrinsic 892
 JPO 899
 lifecycle check 895
 process events 912
 program range 896
 relationship events 909
 state events 910
 wizard 895
 workflow 899
mail. *See* IconMail.
maintaining stores 140
map file
 compare command 228
 import command 219
margins
 form 590
 report 570
matchlist comparison 797
matrix.ini file
 variables
 MX_DECIMAL_SYSTEM 85
 MX_TIMER_TRACE 96
 MX_VERBOSE_PARAM_TRACE 106
matrix.ini file. *See* initialization file.

MatrixLogWriter
 method 99
 tracing with 103
maximum
 on a query 814
 on a set 781
median
 on a query 815
 on a set 781
memory
 controlling requirements for queries 809
 monitoring 109
menu
 adding 647
 alt 648
 command 649
 copying 650
 defining 647
 deleting 653
 description 647
 icon 647
 label 648
 list 650
 menu 648
 modifying 650
 name 647
 property 649
 ref 648
 setting 649
messages
 sending 675
 sending based on policy 396
method
 assigning to type 329
migrating
 databases 223
 files 223
 revision chains 224
MIME type
 format 368
 page 619
 resource 628
minimum
 on a query 814
 on a set 781
minus operator 793
modal dialogs 497
modeling considerations 811
modified date, select clause 60
modify access 251
 described 251
 for attribute rules 376

for relationship rule 377
modify alias 612
modify association 302, 302
modify attribute 320
modify businessobject 712
 add history clause 753
 introduced 711
modify command 639
modify connection 359, 360
modify cue 841, 842
modify filter 831, 832
modify form 597
modify format 370
modify inquiry 660
modify location 155, 160, 176
modify menu 650
modify page 621
modify person 284
modify policy 408
 introduced 408
 signature subclauses 413
 state subclauses 411
modify process 555
modify program 450, 510
modify property 606
modify query 820
modify relationship 354
 from subclauses 356
 introduced 354
 to subclauses 356
modify report 582
modify resource 630
modify rule 379
modify server
 copy clause 179
 introduced 176
 link clause 178
 master clause 178
modify set 770
modify statement 58
modify store
 description 138
 filename hashed 138
 host 138
 icon 138
 introduced 138
 lock 138
 name 138
 path 138
 permission 138
 unlock 138
modify table 871
modify tip 851
 clauses 852
 introduced 851
modify toolset 859, 860
modify type 189, 332
modify vault
 description 118
 hidden 118
 icon 118
 introduced 118
 name 118
 property 118
modify view 885
modify wizard
 frame subclauses 452
 introduced 450
 widget subclauses 452
modify workflow 688
modifyform access
 described 255
 for form rules 376
modifying
 association definition 302
 attribute definition 320
 business object definition 711
 business object state 746
 command definition 639
 connection attributes 360
 databases 56
 filter 831
 form definition 597
 format definition 370
 group definition 295
 inquiry definition 660
 items 58
 location definition 155, 160
 menu definition 650
 page object definition 621
 person definition 284
 policy definition 408
 policy states 411
 process definition 555
 program definition 450, 510
 query definition 820
 relationship connection ends 356
 relationship definition 354
 relationship instances 359
 report definition 582
 resource object definition 630
 role definition 295
 rule definition 379
 server definition 176

set definition 770
 signature requirements 413, 746
 store 138
 table definition 870
 tip 851
 toolset 859
 type definition 332
 vault definitions 118
 view 885
 visual cue definition 841
 widgets 452
 wizard frames 452
 workflow definition 688
monitor context command 107
monitoring
 context objects 107
 memory 109
MQL
 accessing 43
 help 51
 interactive mode 36
 programs 461
 quitting 50
 script 47
 script mode 36
 shell command 488, 509
 statements 47
 syntax 48
 Tcl mode 39
 tracing 95, 97
 using 36, 53
 window, suppressing 45
mq1
 -b option 44
 -c option 45
 command options 43
 -d option 45
 -k option 45
 -stderr option 46
 -stdin option 45
 -stdout option 45
 -t option 45
 multiline widget 435
 multiplication operator 793
multi-trigger event
 create events 523
 relationship events 524
MX_ALIASNAME_PREFERENCE 611
MX_CURRENT_TIME 794
MX_DECIMAL_SYSTEM 85
MX_FTP_TRACE 96
MX_LANGUAGE_PREFERENCE 611
MX_LDAP_TRACE 96
MX_MQL_TRACE 96, 476
MX_SITE_PREFERENCE 158
MX_SMTP_TRACE 96
MX_SQL_TRACE 96
MX_TIMER_TRACE 96
MX_TRIGGER_RECUSION_DETECTION
 setting 535
MX_TRIGGER_RECUSION_LIMIT
 setting 535
MX_TRIGGER_TRACE 95
MX_VERBOSE_PARAM_TRACE 106
MX_VERBOSE_TRACE 96
MX_WORKFLOW_TRACE 96
mxtrace.log 90

N

name macro 643
 name mangling 899
 naming a business object 695
needsbusinessobject 462
 nesting programs
 introduced 477
 rules 477
 Netscape Directory Server 263
nfs
 in add location statement 132, 151
NLS_LANG 85
 non-abstract types 325
 none revision/clone rule 349
 non-indexed fields 809
notice 477
 notification commands 477
 NT event viewer 94

O

object ID 697
 object tip. *See* tip.
 object type, assigning attributes 308
 observer program
 wizard frame definition 433
OID. *See* object ID
openedit businessobject 741
openLDAP 263
openldap.org 263
openview businessobject 741
 option, in statements 50
Oracle 85
 alias 170
 constraints 84

decimal setting 85
instance and eMatrix server 171
Oracle error
 ORA-00987
 missing or invalid usernames 170
order
 table column 868
order when creating definitions 55
output 91
 from programs 508
 programs 499
 redirecting 45
 Tcl format 68
 to file 45
output 53
output length
 using in programs 485
override access 254
override businessobject 749
override trigger 522
owner
 of business object 701
owner access
 assigning in policy 397
 defined 244
 policy 386
ownership, reassigning in policy 397

P

page
 adding 617
 defined 616
 deleting 622
 description 618
 file 617, 618
 hidden 619
 icon 618
 mime 619
 modifying 621
 name 617
 property 619
 supporting different and formats 623
parent program 502
parentheses in where clause 792
password 671
 allow reuse 266
 allow username 266
 changing 670
 changing context 506
 cipher 266
 context 669
 disabling 278
 encrypting 267
 expired 278
 expires 265
 FTP 134
 indicating none 278
 lockout 265
 maximum size 265
 minimum size 265
 mixed alphanumeric 266
 system-wide settings 264
 widget 435
password 53
pattern operator in attribute ranges 314
performance
 improving 186
 index select 191
 setting history log off 86
permission
 owner, group, world 133, 152
 read, write, execute 133, 152
person
 adding 268
 address 271
 application user 280
 assigning groups and roles 271
 business administrator 281
 certificate 275
 comment 274
 copying 284
 deleting 287
 describing 274
 disabling e-mail 276
 disabling password 278
 enabling e-mail 275
 enabling IconMail 276
 fax number 276
 full name 276
 full user 280
 hidden 277
 icon 277
 inactive 281
 modifying 284
 name 268
 no password 278
 password expired 278
 phone number 279
 property 279
 site 279
 system administrator 281
 trusted 281
 types 280

vault 277
picas 426, 566, 587, 865
piped programs 478
plus operator 793, 794
points 426, 566, 587, 865
policy
 access items 399
 access precedence 248
 adding 388
 assigning access 397
 change object ownership 397
 changing states 386
 changing store 142
 checkins in state 404
 checkout history for state 405
 copying 408
 default format 391
 defined 384
 defining 388
 defining states 385
 deleting 415
 description 388
 event triggers 403
 format 390
 general behavior 384
 hidden 406
 icon 389
 icon for state 405
 in business object definition 701
 lifecycle 384
 modifying 408
 modifying states 411
 name 388
 number of states 385
 promotion action 395
 promotion check 396
 promotion notification 396
 property 406
 revision sequence 391
 revisionable 386
 revisions in state 404
 state signature 401
 states 385, 394
 store 405
 test for promotion 405
 type 389
 user access 386
 versionable 386
port clause
 location 151
port command
 store 132
position, widget 434
Power Web. See Web version.
print 61
print business object
 Tcl command 59, 709
print businessobject
 dump 709
 introduced 705
 output 710
 sortattributes 710
print connection 360
print form 600
print set
 Tcl command 778
print set
 dump clause 778
 introduced 777
 output clause 779
 recordseparator clause 778
 select 778
 selectable 777
print table 876
print transaction 74
printing
 business object field names 708
 business object information 705
 form 600
 history 754
 items 61
 relationship connections 360
 selectable fields 777
 set information 777
 system settings 87
 table 876
 vault 121
printing conventions xxv
privileged business administrator 86
process
 adding 543
 and 552
 attribute 544
 automated 548
 attribute 549
 description 549
 program 550
 trigger 551
 user 549
 xcoord 550
 ycoord 550
 autostart 553
 copying 555
 defining 543

deleting 561
description 543
event triggers 554
finish 552
graph
 transition conditions 559
hidden 553
icon 553
interactive
 attribute 545
 description 545
 duration 546
 introduced 544
 priority 546
 rate 547
 trigger 548
 user 545
wizard 546
worklist 547
xcoord 547
ycoord 547
modifying 555
or 552
property 553
start 552
stop 553
subprocess 551
trigger 554
trouble creating 562
validating 562
processing scripts 56
program
 action example 459
 action trigger 528
 adding 458
 attribute range 472
 chaining 502
 check example 460
 check trigger 526
 code 458
 command syntax 470, 915
 copying 510
 defined 457
 defining 458
 deleting 513
 description 461
 dismissing an appl dialog 493
 downloadable 463
 eMatrixApplet windows disabled 500
 threads share context 500
execute 463
external 461
file containing code 462
for execution as needed 460
for LCD environments 484
format definition example 458
hidden 464
icon 312, 462, 589
invoking a wizard from 448
java 461
launching from appl dialog 493
macro processing in 470, 915
macros 421
methods 421
modifying 450, 510
MQL 461
need for business object 462
nesting 477
observer in wizards 433
output 508
output length 485
override trigger 527
piped option 464
property 465
range values 315
rule 462
select expressions 481
selectable fields 481
table column 868
Tcl on the Web 503
transactions 473
type 461
uploading and downloading files 444
use of backslash 470, 915
usesexternalinterface 463
program environment
 variables 437
program range macros 896
programming on the Web 500
prologue for frame 428
promote access 253
promote businessobject 750
promotion
 notification 396
 of state 395
 signature 401
 tested in state 405
properties
 importing 214
 inherited 325
property
 adding 605
 listing 606
 modifying 606

selecting 607

protocol

ftp 132, 151

nfs 132, 151

protocol clause

location 151

protocol command

store 132

public access

assigning in policy 397

defined 244

policy 386

purge location 157

purge store 144

Q

-q command option 44

query

adding 789

arithmetic expressions 793

average 814

Boolean expressions 798

Boolean operators 799

business objects 790

comparative expressions 792

complex Boolean expressions 800

correlation 815

count 814

date/time 811

defined 786

defining 789

deleting 823

evaluating 821

expandtype 791

hidden 791

improving performance 119

in LCD environments 484

indexed fields 808

keywords 812

maximum 814

median 815

memory requirements 809

minimum 814

modeling considerations 811

modifying 820

name 789

never use only non-indexed fields 809

non-indexed fields 809

optimizing 808, 809

owner 791

query expression 792

relational operators 795

reserved words 812

saving results as set 821

select expressions 812

select fields 808

standard deviation 815

strategies 808

sum 814

temporary 787

unexpected results 812

using defaults 789

using reserved words 812

vault 791

where clause 792

wildcard characters 810

quit 50

quitting MQL 50

quotes

and apostrophes 50

command option 44

double 50

in statements 50

single 50

using mixed double and single 817

R

range

table column 868

range program 315

read access 251

described 251

for attribute rules 376

more about 255

reading 93

reassign

workflow activity 558

reassign set 496

reassign workflow 686

recovery plan 110

recurse, expand businessobject 731

recursion 535

refreshing browsers 495

rehash store 141

reject businessobject 747

relational operators

attribute ranges 312

in queries 795

relationship

adding 341

assigning attributes 308

attribute 341

between business object and set 805
cardinality 346
clone rule 351
connection end modifications 356
copying 354
defined 338
defining 341
deleting 358
description 343
display attributes 496
display history 496
FROM connection 344
hidden 353
icon 343
IDs 359
instances 359
meaning 346
modifying 354
name 341
preventDuplicates 352
propagate connection 352
propagate modify 351
property 353
revision rule 348
rule
 float 350
 none 349
 replicate 350
timestamp 352
TO connection 344
triggers 524
type 345
relationship instances
 deleting connections 360
 freezing connections 360
 introduction 359
 modifying 359
 modifying connection attributes 360
 printing connections 360
 thawing connections 360
remote vault 115
renumbering IconMail messages 678
replicate revision/clone rule 350
replicating captured stores 148
report
 adding 566
 compare command 229
 copying 582
 defined 564
 defining 566
 defining appearance 574
 deleting 584
description 567
design 565
dividing rule 571
evaluating 581
field definition 572
field label 577
field types 571
fields 571
footer 569
header 568
hidden 580
icon 568
layout 565
margins 570
modifying 582
name 566
property 580
size 568
units 566
reserved words in queries 812
resource
 adding 627
 copying 629
 defined 626
 deleting 631
 description 627
 file 627
 hidden 628
 icon 628
 MIME type 628
 modifying 630
 name 627
 property 628
 supporting different formats 632
resuming workflow 685
retrieving
 business object image 700
 workflow image 682
return command 500
revise access 253
revise businessobject 718
revision designator for business object 696
revision macro 643
revision rule
 assigning to relationship 348
 float 348
 none 348
 replicate 348
revision sequence 391
revisionable, state 386
revisions
 allowed in state 404

business object 718
migrating 224
revoke access 253
revoking access 715
RMI
 memory statistics 109
role
 assigning to a person 272
 defining 291
 deleting definition 295
 hierarchy 290
 modifying definition 295
 multiple parents 291
 name 291
 property 295
RPE
 and MQL 467
 commands 467
 downloadable programs 500, 502
 global variables 468
 introduction 420
 local variables 468
 program variables 469
rule
 access precedence 248, 375
 adding 375
 cloning 379
 copying 379
 deleting 382
 description 375
 hidden 376
 icon 375
 introduced 245
 modifying 379
 name 375
 owner access 377
 property 376
 public access 377
 user access 377
run statement 53
running scripts 53
runtime program environment. *See* RPE.

creating baseline for 225
names 183
script
 abort on error 45
 building 53
 comments 49
 comments in 54
 creating definitions 55
 example 47
 processing 56
 running 53
 startup 45
 writing 53
 writing second 56
script mode
 advantages 36
 defined 36
scroll widget 435
second script 56
Secure FTP
 See also FTP
 enabling for captured store 136
See also event triggers
select
 index 191
select businessobject 705
select expression macros 643
select expressions
 in exact equal comparisons 812
 in queries 812
 syntax 802
select fields in queries 808
select history 754
select node 686
select output, Tcl format 68
selectable fields
 for admin objects 60
 for business object 706
 for programs 481
 for wizards 481
 fromset 805
 toset 805
send mail
 cc clause 676
 introduced 675
 subject clause 676
 text clause 676
 to clause 675
sending IconMail 675
server
 connect string 170, 172
 copy 179

S

scale
 table column 868
schedule access 252
scheduled date for state 702
schema
 comparing 225
 comparing with baseline 226

creating 171
defining 171
deleting 177
description 172
diagnostic tools 95
disabling 177, 180
hidden 175
icon 172
importing 213
link 178
master 178
Oracle alias 170
parameters 171
password 172
property 175
timezone 172
user 172

session
monitoring 105
tracing 95

sessions command 90

set
adding 767
average 781
business object collection 768
checkout 496
copying 770
correlation 782
count 780
defined 766
defining 767
deleting 783
dump clause 776
expand relationship 774
expand type 774
expanding from 772
expanding object connections 772
expanding objects 772
expanding to 773
expressions 780
hidden 768
maximum 781
median 781
member 767
minimum 781
modifying 770
name 767
printing 777
property 768
reassign 496
relationship expressions 772
saving expansion information 775

saving query results as 821
selecting fields to print 777
send via iconmail 497
standard deviation 781
sum 780
terse clause 776
viewing definition 777
vs. connection 766

set checkshowaccess 257
set context 669
set password
allowreuse clause 266
allowusername clause 266
cipher clause 266
expires clause 265
lockout clause 265
maxsize clause 265
minsize clause 265
mixedalphanumeric clause 266

set system 83
set transaction 74
set workspace 884
setting
table column 868
setting the workspace 305
shell command 488, 509
and macro processing 470, 915
described 53

show access
described 255
more about 256
setting the global flag 257

signature
approve 401, 746
defining with associations 298
finding date of 756
ignore 401, 747
introduced 401
modifying 413
modifying requirements 746
override requirement 749
promotion 401
reject 401, 747
unsign 748

single quotes
using with double quotes 817

Single Signon 263

site
defined 148
defining 158
deleting 161
description 158

hidden 159
icon 158
member 159
property 159
Siteminder 263
size
 form 591
 frame 427
 report 568
 table column 868
 widget 434
smatchlist comparison 797
sort
 table column 868
source command 500
spaces in strings 440
special characters in Tk 41
splash screen, suppressing on startup 45
SQL tracing 95
standard deviation
 on a query 815
 on a set 781
start subclause for widget 434
start transaction 74
startup options 44
state
 changing 386
 checkout history 405
 defining 385
 file checkin allowed 404
 icon 405
 modifying 411
 modifying signature requirements 413
 name 394
 number required 385
 policy 385
 promotion 395
 reassigning ownership 397
 revisionable 386
 revisions allowed 404
 scheduled date in business object
 definition 702
 signature 401
 test for promotion 405
 versionable 386
statements
 basic 53
 clauses 49
 components 49
 diagram 49
 keyword 50
 MQL 47
options 50
values 49
writing 48
statistics
 memory for RMI server 109
object 107
-stderr 46
-stdin 45
-stdout 45
stdout 95
store
 access privileges 133
 captured files 125
 changing 149
 data tablespace 135
 defined 124
 defining 127
 deleting 144
 description 128
 disk space 142
 file checkin 740
 filename hashed 129
 files within 124
 fragmented files 140
 FTP password 134
 FTP username 134
 hidden 135
 host 131
 icon 128
 index tablespace 135
 ingested 125
 inventory 140
 listing contents 140
 lock 131
 maintaining 140
 modifying 138
 name 127
 overriding on file checkin 740
 path 128, 131
 permission 132
 port 132
 property 135
 protocol 132
 purge 144
 rehash 141
 replacing 149
 too full 149
 tracked files 126
 type 129
 validate 142
 string concatenation 794
 structure

copying 737
 deleting 737
 listing 737
 printing 737
sum
 on a query 814
 on a set 780
suspending
 programs 497
 workflow 685
sync on demand 162
synchronizing captured stores 162
synchronous data replication 178
syntax
 clauses 49
 comments 49
 general 57
 keywords 50
 MQL 48
 options 50
 quotes 50, 817
 rules 48
 statements 48
Tcl 41
 values 49, 50
system administrator 281
 responsibilities 82
system decimal symbol 85
system settings
 listing 87
 printing 87
system-wide settings 83

T
-t command option 45
table
 active 865, 881
 adding 865
 column alt 868
 column definition 867
 column edit 869
 column height 869
 column hidden 869
 column href 868
 column minsize 868
 column order 868
 column program 868
 column range 868
 column scale 868
 column setting 868
 column size 868

column sorttype 868
 column update 868
 column user access 868
 column width 869
 columns 867
 copying 870
 defining 865
 deleting 875
 description 866
 hidden 869
 icon 867
 list 870
 modifying 870
 name 865
 printing 876
 property 869
 units 865
tablespace
 data 116, 135
 index 116
 naming 116, 135

Tcl
 and escape characters 819
 and the Web 503
 clause 68
 clause used in expand command 736
 clause used in expand set command 776
 clause used in list admintype command 61
 clause used in print command 59, 709
 clause used in print set command 778
 command syntax 41
 exec command 488, 509
 exit command 500
 format for select output 68
 mode 39
 output 508
 parsing backslashes 471
 return 1 command 491
 return command 500
 source command 500
 special characters 41
 tracing 95, 103
temporary query 787
thaw access
 described 253
 for relationship rule 376
thaw connection 360
threads share context 500
tidy
 system setting 86
tidy store 140
tidy vault 120

time 794
 tip
 active 847
 applies to 847
 copying 851
 deleting 853
 expression clause 849
 hidden 849
 list 851
 modifying definition 851
 name 847
 order 849
 owner 849
 property 850
 query expressions 849
 where clause 849
 Tk 41, 493
 toconnect access
 described 255
 for relationship rule 376
 todisconnect access
 described 255
 for relationship rule 376
 Tool command language 39
 toolset
 active 857
 copying 859
 deleting 861
 hidden 857
 modifying definition 859
 name 857
 property 857
 toset selectable for business objects 805
 tracing 95
 ADK call parameters 106
 ADK interfaces 98
 all-session vs. single session 95
 and debugging 101
 Context.printTrace() 104
 Context.setTrace() 104
 MQL 97
 with MatrixLogWriter 103
 with Tcl 103
 tracked
 file 126
 store 125
 transaction
 control 74
 explicit 74
 implicit 74
 program 473
 statements 74
 transaction boundaries
 event triggers 518
 extending 74
 transition condition 559
 trigger 539
 trigger off 190, 540
 trigger on 540
 trusted user 281
 type
 abstract 325, 329
 adding 187, 326
 attribute 188, 327
 characteristics 325
 copying 332
 creating a business object 695
 defined 324
 defining 187, 326
 derived 328
 description 187, 326
 explicit characteristics 325
 form 329
 hidden 330
 icon 187, 327
 implicit characteristics 325
 inherited properties 325
 inheriting attributes 328
 method 329
 modifying 332
 name 187, 326
 non-abstract 325
 property 188, 331
 trigger 330
 type macro 643

U

UNC paths 470, 915
 units
 frame 426
 inches 426, 566, 587, 865
 picas 426, 566, 587, 865
 points 426, 566, 587, 865
 UNIX 93
 unknown, program value of 799
 unlock access
 described 252
 more about 256
 unlock businessobject 745
 unsign businessobject signature 748
 update
 table column 868
 update set 83

updating browsers 495
 upload
 within a wizard 435
 upload command 444
 URL access 504
 user
 authenticating with LDAP 263
 creator 268
 guest 268
 user access
 assigning in policy 397
 assigning in rule 244
 policy 386
 table column 868
 user name
 changing context 506
 context 669, 671
 using
 MQL 36
 Tcl 39
 using NT viewer 94

V

-v command option 45
 validate 88
 validate index vault 119
 validate process 562
 validate program, widget 432
 validate store 142
 validate vault 88
 validation
 database 88
 index 191
 levels 88
 value
 defined 49
 syntax 50
 vault
 administration 113
 business object 113
 clearing 119
 data tablespace 116
 defined 113
 defining 114
 deleting 122
 description 114
 file 116
 fixing fragmented 120
 foreign 115
 hidden 117
 icon 115

index tablespace 116
 indexing 119
 interface 116
 library path 116
 local 115
 map 117
 map file name 116
 modifying 118
 name 114
 print 121
 property 117
 remote 115
 server 116
 specifying to optimize queries 808
 tidy 120
 types 113, 115
 validating 88
 working with 112
 verbose
 compare command 230
 mode 45
 verbose
 introduced 54
 version 54
 versionable, state 386
 view
 active 885
 copying definition 883
 deleting 887
 hidden 881
 modifying definition 885
 name 881
 property 881
 specifying in appl navigator commands 495
 viewform access
 described 255
 for form rules 376
 viewing
 business object definition 705
 current users 90
 set definition 777
 visual cue
 active 838
 adding 837
 applies to 838
 color 839
 copying 841
 deleting 844
 font 839
 hidden 840
 highlight color 839
 line style 839

listing 841
modifying 841
name 837
order 839
owner 840
property 840
query expressions 840
vault 839
where clause 840

visuals
copying 304
deleting 305

W

warning 477
Web
consideration for designing forms 591, 593
programming 500
web form 587
where clause
on business objects 733
on cues 840
on queries 792
on tips 849
query strategies 808
recommendations 792
using parentheses 792

widget
autoheight 434
autowidth 434
color 433
complex 440
definition 419
drawborder 435
edit 435
font 434
load 432
modifying 452
multiline 435
name 431
password 435
position 434
scroll 435
size 434
upload 435
validate 432
value 431

wildcard
in queries 810
limiting to optimize queries 808

wizard

adding 422
and appl dialogs 448
code 422
colors in frame 427
command line interface 447
copying 450
creating 422
definition 418
deleting 454
description 423
downloadable 424
epilogue 429
eval statements 443
execute 424
file 423
frame
sequence 439
skipping 428
frames 425
hidden 425
icon 423
icon in frame 430
introduced 418
invoking from a program 448
linking 448
macros 440, 895
modifying 450
modifying frames 452
modifying widgets 452
name 422
need for business object 423
observer program 433
programming strategy 436
prologue 428
property 425
running 447
 using eMatrix 447
selectable fields 481
sequence of functions 438
size of frame 427
skipping frames 428
spaces in strings 440
status in frame 429
testing 447
type 423
units in frame 426
uploading and downloading files 444
widgets in frame 430

workflow
adding 681
assigning activities to multiple users 689
attribute 683

- defining 681
- deleting 691
- description 681
- execute automated 550
- exporting 203
- image 681
- importing 213, 221
- modifying 688
- owner 683
- planning execution 686
- reassigning 686
- reassigning activity to a group 558
- resuming 685
- retrieving image 682
- retrieving image files 682
- starting 685
- stopping 685
- suspending 685
- vault 683

workflow instances, not analyzed in compare 226

workflow macros 899

workspace

- importing 213
- setting 305, 884

writing

- scripts 53
- statements 48

X

- XML
 - export 204
 - import 209
 - using for schema comparison 225
- xml clause 205
- XML Exchange
 - port 132, 151
 - protocol 132, 151
- xml statement 205