Progress®



OpenEdge Programming Interfaces

OpenEdge®

Copyright

© 2021 Progress Software Corporation and/or its subsidiaries or affiliates. All rights reserved.

These materials and all Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. The information in these materials is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear therein. The references in these materials to specific platforms supported are subject to change.

DataDirect Cloud, DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, DataRPM, Defrag This, Deliver More Than Expected, DevReach (and design), Icenium, Inspec, Ipswitch, iMacros, Kendo UI, Kinvey, MessageWay, MOVEit, NativeChat, NativeScript, OpenEdge, Powered by Chef, Powered by Progress, Progress, Progress Software Developers Network, SequeLink, Sitefinity (and Design), Sitefinity, Sitefinity (and design), SpeedScript, Stylus Studio, Stylized Design (Arrow/3D Box logo), Styleized Design (C Chef logo), Stylized Design of Samurai, TeamPulse, Telerik, Telerik (and design), Test Studio, WebSpeed, WhatsConfigured, WhatsConnected, WhatsUp, and WS_FTP are registered trademarks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and/or other countries.

Analytics360, AppServer, BusinessEdge, Chef Automate, Chef Compliance, Chef Desktop, Chef Habitat, Chef WorkStation, Corticon.js, Corticon Rules, Data Access, DataDirect Autonomous REST Connector, DataDirect Spy, DevCraft, Fiddler, Fiddler Everywhere, FiddlerCap, FiddlerCore, FiddlerScript, Hybrid Data Pipeline, iMail, JustAssembly, JustDecompile, JustMock, KendoReact, NativeScript Sidekick, OpenAccess, PASOE, Pro2, ProDataSet, Progress Results, Progress Software, ProVision, PSE Pro, Push Jobs, SafeSpaceVR, Sitefinity Cloud, Sitefinity CMS, Sitefinity Digital Experience Cloud, Sitefinity Feather, Sitefinity Insight, Sitefinity Thunder, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, Supermarket, SupportLink, Unite UX, and WebClient are trademarks or service marks of Progress Software Corporation and/or its subsidiaries or affiliates in the U.S. and other countries. Java is a registered trademark of Oracle and/or its affiliates. Any other marks contained herein may be trademarks of their respective owners.

July 2021

Product version: Progress OpenEdge 12.4

Updated: 2021/06/28

Table of Contents

Part I: Data Management....33

Database Access....35

Database connections....36
- Connection parameters....36
- Secure database connections....36
- The CONNECT statement....37
- Auto-connect....39
- Multi-database connection considerations....39
- Run-time connection considerations....40
- Connection failures and disruptions....40
- OpenEdge database connection functions....41
- Balancing conserving connections with minimizing overhead....42

Disconnecting databases....43
Logical database names....43
Database aliases....44
- Creating aliases in applications....45
- Compiling procedures with aliases....46
- Using shared record buffers with aliases....47

Data-handling statements....47
Adding and deleting records....49
Defining a set of records to fetch....49
Fetching records....50
- Order of fetched records....51
- Sample record fetches....51
- ROWID and RECID data types....52
- Results lists....54
- FIND repositioning....58

Fetching field lists....59
- Field list benefits....59
- Specifying field lists in ABL....59
- Avoiding implied field list entries....61
- Updating and deleting with field lists....61
- Cursor repositioning and field lists....62
- Field list handling in degenerate cases....62

Joining tables....63
- Specifying joins in ABL....63
- Using inner joins....66

Using left outer joins....67
Implementing other outer joins....68
Mixing inner and left outer joins....68
Server-side join processing....68
The CONTAINS operator....71
The CONTAINS operator and word indexes....72
Word indexes and word-break tables....72
Creating and viewing word indexes....72
Using word-break tables....72
Writing queries using the CONTAINS operator....78
Word indexing external documents....80
Word indexing non-OpenEdge databases....81
Sequences....81
Choosing between sequences and control tables....82
Creating and maintaining sequences....85
Accessing and incrementing sequences....85
Database trigger considerations....89
Storing triggers in an operating system directory....90
Storing triggers in an r-code library....90
Using the RAW data type....90
Multi-database programming techniques....91
Referencing tables and fields....91
Positioning database references in an application....93
Using the LIKE option....93
Creating schema cache files....93
Building a schema cache file....94
Example schema cache file creation....94
Using a schema cache file....96

Application Security 97

Types of application security....97
Elements of an application authentication model....98
Authentication systems....98
Security domains....99
Domain registries....99
User credentials....99
Security tokens (client-principal objects)....99
User identity management mechanisms....100
Client context management mechanisms....100
Application authentication design options....100
Elements for authenticating ABL user identity....101
Authentication systems supported in ABL....101
Specifying user IDs and passwords in ABL....102
Password encryption....103

Security tokens in ABL....103
Authenticating and managing user identity in ABL....104
User identity applications....105
ABL for managing user identity....106
Client-principal objects 111
Setting up and using domain registries....123
Authenticating and setting user identity....128
Establishing database connection identity 135
Establishing ABL session identity....141
Establishing and managing identity for multi-tier applications....145
Establishing auditing identity....154
Creating and managing unique object identities 154
Authorizing access to procedures and database resources....155
Authorizing users from within a given procedure....156
Defining activities-based user authorization....159
Authorizing user access to tables and fields....165
Using ABL callbacks in authentication systems....166
Advantages of using ABL authentication callbacks....166
Running ABL callbacks....166
Adding an ABL callback to an authentication system....167
Extending the built-in systems where OpenEdge performs user authentication....167
Extending SSO-only authentication systems....168
Using ABL to implement user authentication systems....169
Creating an ABL authentication callback procedure....171
Using cryptography to secure data....175
Creating and maintaining a cryptography policy....176
Generating encryption keys....178
Implementing symmetric cryptography in ABL....181
Using message digests in ABL....185
Managing and transporting crypto data....186

Multi-tenant ABL....189

Running applications in a multi-tenant database environment....190
Existing applications....190
Super-tenant applications....190
Coding for super-tenant access....190
Setting effective tenancy....191
Tenancy identification....192
Super-tenant queries....194
Super-tenant data updates....199
Super-tenant sequence access....200
Using multi-tenant-enabled sequences....200
Using multi-tenant sequences with shared and group-configured tables....201
Restrictions for using groups and sequences....202

Managing access to tables with groups....202

ABL API for Multi-tenant and Table Partition Management....205

Overview of the API....206
API packages, classes, and interfaces....207
API entities and entity service methods....208
Instantiating a DataAdminService....213
Managing the service life cycle....214
Managing transactions....214
Table updates that require two transactions....215
Maintaining multi-tenant and table partition entities....215
Areas (creating and reading)....216
Authentication systems (creating, reading, updating and deleting)....217
Domains (creating, reading, updating and deleting)....218
Schema (reading, loading, and updating)....219
Sequences (creating, reading, updating and deleting)....220
Tables (reading and updating)....220
Table permissions (reading and updating)....221
Tenants (creating, reading, updating and deleting)....222
Tenant groups (creating, reading, updating and deleting)....223
Users (creating, reading, updating and deleting)....224
User permissions (reading)....225
Partitions (creating, reading, updating, deleting, enabling and allocating)....225
Working with partitions....226
Filtering and sorting....227
Filter expressions....227
Filtering on a child collection using the RequestInfo class....228
Exporting and importing....230
Exporting and importing objects as JSON files....230
Exporting tenants and tenant groups as ABL code....231
Exporting and Importing Partition Policies and Details....232
Executing DataAdminService utilities....232
IDataAdminUtility interface....232
DeallocateUtility class....233
PartitionMergeUtility class....233
PartitionSplitUtility class....235
Handling errors....238
Handling data errors with the DataAdminErrorHandler class....239
Controlling data errors with the IDataAdminService interface....239

Auditing....241

Setting up application security for auditing....241
Configuring the auditing identity....242

Asserting the auditing identity....243
Verifying that auditing is enabled....245
Configuring auditing privileges....245
Recording application events....246
Planning application events....246
ABL for generating application events....247
Instructing OpenEdge to record an application event....248
Managing audit event context....248
Application design considerations....253
Developing auditing tools....255
Custom audit configuration tools....255
Custom audit archiving tools....257

Part II: Input/Output Processes....261

Handling User Input....263
The keyboard and the mouse....264
Key mapping....264
Key monitoring....264
Key translation functions....265
Key codes and key labels....266
Key functions....277
Changing the function of a key....282
Using mouse buttons and events....284
Portable and physical buttons....284
Specifying mouse events....285
Telling the AVM how to continue processing....286
Monitoring keystrokes during data entry....289
Editing blocks....290
User interface triggers....290
Using editing blocks and user interface triggers....290

Alternate I/O Sources....297
Understanding input and output....298
Changing the output destination 298
Output to printers 299
Additional options for printing in Windows....299
Output to files or devices 301
Output to the clipboard 301
Resetting output to the terminal 301
Sending output to multiple destinations 302
Stream I/O vs. screen I/O....303
A printing solution 304

Changing the input source of a procedure....306
Input from files....306
Input from devices, directories, and the terminal....307
Receiving input from multiple sources....307
Reading input with control characters....309
Defining additional input/output streams....309
Sharing streams among procedures....311
Defining a shared stream....311
Using stream object handles....312
Summary of opening and closing streams....313
Processes as input and output streams (NT and UNIX only)....314
Portability issues....315
I/O redirection for batch jobs....315
Reading the contents of a directory....315
Performing code-page conversions....317
Converting nonstandard input files....318
Importing and exporting data....319

Colors and Fonts....325

Making colors and fonts available to an application....326
OpenEdge default colors....326
OpenEdge default fonts....327
Assigning colors and fonts to a widget....327
Assigning colors and fonts to ActiveX Automation objects and ActiveX controls....329
Color and font inheritance....329
Overriding default color inheritance....329
Color in character interfaces....330
Widget states and color....330
Color specification....330
Colors in windows in graphical interfaces....331
Systems with the Palette Manager....331
Systems without the Palette Manager....331
Managing colors and fonts in graphical applications....331
Allowing the user to change colors and fonts....332
Establishing dynamic colors....332
Color dialog box....333
Color dialog box example....333
Saving a modified color....335
Font dialog box....335
Font dialog box example....335
Saving a modified font....337
Accessing the current color and font tables....337
COLOR-TABLE handle....337
FONT-TABLE handle....338

Retrieving and changing color and font definitions....339
Changing resource definitions....339
Using GET-KEY-VALUE and PUT-KEY-VALUE....340
Managing application environments....340
Understanding environment management....341
Using environment management statements....341
Managing multiple environments....341
Managing color display limitations....342

Creating Reports....343

Report basics....344
Down frames....344
Text widgets....345
Control blocks and output statements....345
Basic report demonstration....345
Designing an interface for viewing report data....347
Converting widget values to report data....350
Printing reports and the STREAM-IO option....350
Formatting long text strings....351
Generating reports with control breaks and aggregates....352
Generating reports from multiple tables....355
Reporting information from one table....356
Reporting information from two tables....356
Reporting information from multiple tables....357
Redirecting output....359
OUTPUT TO and the default stream....359
Directing output to a file....360
Directing output to a printer....362
Directing output to multiple destinations....362
Designing frames for reports....363
Using base fields with the @ option....363
Using the HEADER of a frame for running page heads....364
Using the HEADER of a frame for running page footers....365
Programming example....365
Using the PUT statement....368
Using PUT for printer control....372
Summary....374
Generating simple reports....374
Redirecting output....374
Generating reports of data represented by widgets....374
Customizing reports....374

Part III: External Program Interfaces....375

Introduction to External Program Interfaces.....413

Using MEMPTR to reference external data.....413
Comparing MEMPTR and RAW data types.....414
Initializing and uninitializing MEMPTR variables.....415
Reading and writing data.....416
Retrieving and storing pointers.....420
Setting byte order.....421
Defining a MEMPTR with Memory Stream classes.....422
System clipboard.....423
The system clipboard and ABL.....423
Requirements for using the CLIPBOARD handle.....424
Named pipes.....424
Named pipes and ABL.....424
Requirements for using named pipes.....424
UNIX shared library and Windows DLL support.....424
Shared libraries and ABL.....425
Requirements for using shared libraries.....425
Windows Dynamic Data Exchange.....426
DDE and ABL.....426
Requirements for using DDE.....427
COM objects: Automation objects and ActiveX controls.....427
COM objects supported in ABL.....427
Support for COM object properties and methods.....429
Support for Automation object events.....430
Support for ActiveX control events.....430
COM object sources.....430
Requirements for using Automation objects.....431
Requirements for using ActiveX controls.....431
Programming requirements.....432
Sockets.....432
Reasons to use sockets.....432
Connection model.....433
Server socket and socket objects.....434
ABL socket event model.....434
Programming requirements.....435

System Clipboard.....437

CLIPBOARD system handle.....437
AVAILABLE-FORMATS Attribute.....438
ITEMS-PER-ROW Attribute.....438
MULTIPLE Attribute.....439
NUM-FORMATS Attribute.....439

TYPE Attribute....439
VALUE attribute....439
Single-item data transfers....439
Enabling and disabling clipboard operations....440
Implementing single-item transfers....441
Single-item transfer example....443
Multiple-item data transfers....445
Widget-based transfers....446
Data-based transfers....446
Multiple-item transfer example....447

Providing Help for OpenEdge Applications....449

Field-level online help....449
Status messages....449
ToolTips....453
Online help systems....456
The SYSTEM-HELP statement....457
Accessing online help from the menu bar....460
Calling online help with a Help button....462
Accessing online help with the help key....463
Quitting help when exiting the application....464

Named Pipes....465

Overview of named pipes with ABL....466
Access from ABL....466
Named pipes and files....466
Uses for named pipes....466
Operational characteristics of named pipes....467
Advantages and disadvantages of named pipes....468
UNIX named pipes....468
Creating a named pipe....468
Deleting a named pipe....469
Accessing a named pipe within ABL....469
UNIX named pipe examples....470
Windows named pipes....476
Accessing Windows named pipes....476
Linking OpenEdge and non-OpenEdge processes using Windows named pipes....476
Building and running the sample Windows named pipes application....477
Coding the ABL program....477
Coding the C program....479
Running the application....482

Shared Library and DLL Support....483

Using shared libraries....484
Accessing a shared library entry point....486
Declaring a shared library for static access....486
Declaring a shared library for dynamic access....488
Passing parameters to a shared library routine....490
Shared library parameter data types....490
Passing arrays as parameters....492
Passing the Unknown value and NULL values....496
Using structure parameters....496
Using MEMPTR variables as parameters....497
DLL routines and ABL widgets (Windows only)....498
Executing a shared library routine....498
Executing a shared library routine for static access....499
Executing a shared library routine for dynamic access....499
Parameter data types....499
INT64 parameter conversions....500
Loading and unloading shared libraries....501
Code samples....501
Examples of static access to a shared library....501
Examples of dynamic access to a shared library....504

Windows Dynamic Data Exchange....507

Introduction to DDE....507
Using the DDE protocol....508
The course of a DDE conversation....508
ABL statements for DDE conversations....508
Structuring a DDE conversation in ABL....509
Conversation hierarchy....509
Defining conversation endpoints: DDE frames....510
DDE-ERROR....511
Other DDE attributes....512
Opening DDE conversations....512
Preparing the server application....512
Defining DDE frames....514
Initiating a conversation....514
Exchanging data in conversations....515
Demand-driven exchanges....515
Event-driven exchanges....516
Closing DDE conversations....518
DDE example....519

Using COM Objects in ABL....523

How COM objects differ from ABL widgets....523

Functionality....524
ABL mechanisms....524
Obtaining access to COM objects....525
Accessing COM object properties and methods....525
Property and method syntax....525
Specifying options for properties and method parameters....531
Restrictions on property and method references....533
Managing COM objects in an application....534
Validating component handles....534
Managing COM object font and color settings....534
Navigating ActiveX collections....536
Managing COM object resources....536
Error handling....537
Locating COM object information on your system....538
Using the COM Object Viewer....538
Accessing Type Libraries....538
Locating objects in the Viewer....539
Viewing methods, properties, and events....540

ActiveX Automation Support....543

Requirements for doing Automation....543
Accessing Automation Servers....544
Option 1: Instantiate Automation object by name....544
Option 2: Connect to top-level named Automation object....545
Option 3: Connect to or instantiate a named Automation object and file....546
Option 4: Connect to or instantiate implied Automation object and file....547
Managing Automation objects....548
Automation event support....549

ActiveX Control Support....551

How ABL supports ActiveX controls....552
An example ActiveX control in ABL....552
How ABL encapsulates an ActiveX control....553
Control-frame attributes and properties....554
Additional control functionality....554
Special considerations for extended controls....556
ActiveX control restrictions....556
Creating a control instance in ABL....556
Understanding design time and runtime properties....556
Setting design time properties....557
Setting the ActiveX control name....557
Setting the control-frame name....557
Orienting a control in the user interface at design time....558

Setting control location....558
Setting control height and width 558
Setting control tab order....559
Defining invisible controls....559
Accessing ActiveX controls at runtime....559
Instantiating the control....559
Accessing the control handle....560
Managing ActiveX controls at runtime....562
Managing tab order and Z order....563
Working with ABL key functions....563
Setting graphical properties of an ActiveX control....563
Releasing control resources....563
Handling events....565
Handling control-frame widget events....565
Handling ActiveX control events....565
Managing external procedure files....569
Programming ActiveX controls in the AppBuilder....570
Creating data definitions for an ActiveX control....570
Instantiating the control....572
Initializing the control....575
Using event procedures....575
Interacting outside of event procedures....577
ActiveX controls and examples installed with ABL....577
Combo box control (CSComboBox)....577
Spin button control (CSSpin)....578
Timer control (PSTimer)....578

Sockets....579

ABL for programming sockets....580
Overview of tasks to implement ABL sockets....585
Implementing an ABL socket server....588
Listening and responding to connection requests....588
Managing the server socket....589
Implementing an ABL socket client....589
Read, writing, and managing sockets on clients and servers....589
Defining and initializing MEMPTR variables....590
Detecting data on a socket....590
Reading data on a socket....591
Writing data on a socket....593
Marshalling and unmarshalling data for socket I/O....594
Managing sockets and their connections....594
Implementing ABL socket security....595
Configuring TLS servers and clients....595
Enabling TLS server connections....596

Connecting an TLS client to an TLS server....597
Invoking socket operations for TLS sessions....598
Examples using ABL sockets....599
Socket example using a method as a callback....605
Host Language Call Interface....375
HLC and OpenEdge....376
Uses of HLC....376
Requirements for using HLC....377
Use the CALL statement....377
Map routine identifiers using PRODSP() 378
Indicate whether the C source code is thread safe....381
Guidelines for making the C source code thread safe....381
HLC files and directories....385
Build an HLC library 385
Object files shipped with OpenEdge....387
Write C functions....388
Top-level C function declaration....388
Return error codes from a top-level function....389
Name C functions....389
C function portability....390
Avoid common HLC errors 390
Memory allocation....390
Data size....391
Use HLC library functions....391
Access ABL data....391
Data type conversion....393
Call an HLC library function....394
Timer services....394
User interrupt handling....395
Use a library function that writes to a shared buffer....395
Pass error codes back to ABL....396
HLC applications on Unix systems....396
Handle raw disk I/O....396
Handle terminal I/O....396
Handle abnormal exits....398
Sample HLC application....398
Run the Tank sample application on Windows....404
Run the Tank sample application on Unix....405
Source code listings....407

Appendix A: COM Object Data Type Mapping....609

Data type conversion strategy....610
Converting COM to ABL data types....610
Converting ABL to COM data types....610

General conversion features and limitations....610
Conversions from COM to ABL data types....610
Conversions from ABL to COM data types....611
ABL to COM data type features and limitations....612
Alternate COM data type names....614

Appendix B: Audit Policy Maintenance APIs....617

API overview....618
Reference to API procedure files....618
Using the generic utility and caching APIs with an application server....619
Generic utility API....620
cancel-import-from-xml procedure....620
check-conflicts procedure....621
export-audit-events procedure....621
export-cached-audit-events procedure....621
fill-audit-dataset procedure....622
getAuditEvents procedure....622
get-complete-dbname-list procedure....623
get-DB-Name function....623
get-dbname-list procedure....624
get-policies-merge procedure....624
has-DB-option function....624
isDBOnAppServer function....625
override-policies-from-xml procedure....626
policies-dataset-read-xml procedure....626
policies-dataset-write-xml procedure....627
setAppServerHandle procedure....627
updateAuditEvent procedure....627
update-auditing-policies procedure....628
Caching API....630
changeAuditDatabase procedure....631
export-cached-audit-events procedure....631
export-policies-to-xml procedure....632
get-audit-events procedure....632
get-conflict-info procedure....632
getDataModified procedure....633
get-merge-info procedure....633
getNextAppLevelEventID function....633
hasApplEvents function....633
import-audit-events procedure....633
import-policies-from-xml procedure....634
is-valid-event-id function....634
registerAuditTableHandle procedure....634
rejectChangesAuditDatabase procedure....635

resubmit-import-from-xml procedure....635
saveAuditEventChanges procedure....635
saveChangesAuditDatabase procedure....635
set-tt-hdls-for-dataset procedure....636
Exporting policies to XML....636
_exp-policies.p....636
_exp-policies-db.p....636
Importing policies from XML....637
copy-changes-to-dset procedure....638
import-xml-db procedure....639
import-xml-fill-dset procedure....639
refresh-db-cache procedure....640
save-changes-to-db procedure....640
Additional audit policy procedures....641
_aud-conflict.p....641
_get-audevents.p....641
_get-db-list.p....642
_get-policies.p....642
_update-policies.p....642

Appendix C: Change Data Capture ABL references....645

API entities and entity service methods....646
New( ) method (Table policy)....646
New( ) method (Table policies)....647
New( ) method (Field policy)....647
Create( ) method....647
Create( ) method (Table policies)....648
Get( ) method....648
Get( ) method (Table policies)....648
GetTable( ) method (Table policy)....649
GetFilter( ) method (Table policies)....649
Update( ) method (Table policy)....649
Update( ) method (Table policies)....650
Delete( ) method....650
Code examples....651
Create CDC policies....651
Create bulk CDC policies....652
Get CDC policies....654
Update CDC policies....655
Delete CDC policies....656
Enums....656
Interface properties (Table policy)....657
Interface properties (Field policy)....660
API to generate Change Data Capture policy program....661

ABL procedure to dump....661
Understanding the structure of a .cd file....662
ABL procedure to load....666
ABL support classes for CDC....667
CDCTrackingHelper class....667
CDCOperation enumeration....668
ABL examples....668

Appendix D: HLC Library Function Reference....671

Function summary....671
Shared-variable access....671
Shared-buffer access....672
Screen display....673
Interrupt handling....673
Timer-service routines....673
Function reference....674
prockint( ) - Test for Interrupt Key....674
proclear( ) - Clear the Display....674
procncel( ) - Cancel Interval Timer....675
proevt( ) - Set Interval Timer....675
profldix( ) - Return Field Handle....676
promsgd( ) - Display Message....676
prordbc( ) - Read CHARACTER Field....677
prordbd( ) - Read DATE Field....678
prordbi( ) - Read INTEGER Field....679
prordbl( ) - Read LOGICAL Field....680
prordbn( ) - Read DECIMAL Field....681
prordbr( ) - Read RECID Field....682
prordc( ) - Read CHARACTER Variable....683
prordd( ) - Read DATE Variable....684
prordi( ) - Read INTEGER Variable....685
prordl( ) - Read LOGICAL Variable....685
prordn( ) - Read DECIMAL Variable....686
prordr( ) - Read RECID Variable....687
prosccls( ) - Set Terminal To Raw Mode and Refresh....688
proscopn( ) - Set Terminal To Initial Mode....689
prosleep( ) - Sleep For Specified Interval....689
prowait( ) - Wait For Timer To Expire....689
prowtbc( ) - Write CHARACTER Field....690
prowtbd( ) - Write DATE Field....691
prowtbi( ) - Write INTEGER Field....692
prowtbl( ) - Write LOGICAL Field....693
prowtbn( ) - Write DECIMAL Field....693
prowtbr( ) - Write RECID Field....694

prowtc( ) - Write CHARACTER Variable....695
prowtd( ) - Write DATE Variable....696
prowti( ) - Write INTEGER Variable....697
prowtl( ) - Write LOGICAL Variable....698
prowtn( ) - Write DECIMAL Variable....699
prowtr( ) - Write RECID Variable....699

Appendix E: Command and Utility Reference....701

QUOTER utility....701

Appendix F: Database Administration Entity Interface Reference....707

IAdministrator interface....709
IAdministrator properties....709
IAdministrator methods....709
IAdministrator examples....709
IArea interface....710
IArea properties....710
IArea methods....711
IArea examples....711
IAuthenticationSystem interface....712
IAuthenticationSystem properties....712
IAuthenticationSystem methods....715
IAuthenticationSystem examples....715
IDataAdminElement interface....716
IDataAdminElement properties....716
IDataAdminElement methods....717
IDataAdminElement examples....717
IDomain interface....718
IDomain properties....718
IDomain methods....720
IDomain examples....720
IExtent interface....721
IExtent properties....721
IExtent methods....722
IExtent examples....722
IField interface....722
IField properties....723
IField methods....723
IFieldPermission interface....723
IFieldPermission properties....723
IFieldPermission methods....724
IFieldPermission examples....724
IIndex interface....725

IIndex properties....725
IIndex methods....726
IIndexField interface....726
IIndexField properties....726
IPartition interface....727
IPartition properties....727
IPartition methods....728
IPartition examples....729
IPartitionOwner interface....729
IPartitionOwner properties....729
IPartitionOwner methods....730
IPartitionParent interface....731
IPartitionParent properties....731
IPartitionPolicy interface....733
IPartitionPolicy properties....733
IPartitionPolicy methods....735
IPartitionPolicy examples....736
IPartitionPolicyDetail interface....739
IPartitionPolicyDetail properties....739
IPartitionPolicyDetail methods....740
IPartitionPolicyDetail examples....741
ISchema interface....744
ISchema properties....744
ISchema methods....746
ISchema examples....746
ISchemaElement interface....746
ISchemaElement properties....747
ISchemaElement methods....747
ISequence interface....748
ISequence properties....748
ISequence methods....749
ISequence examples....749
ISequenceValue interface....749
ISequenceValue properties....750
ISequenceValue methods....750
ISequenceValue examples....750
ITable interface....751
ITable properties....751
ITable methods....752
ITable examples....752
ITablePermission interface....754
ITablePermission properties....754
ITablePermission methods....755
ITablePermission examples....755
ITenant interface....755

ITenant properties....756
ITenant methods....758
ITenant examples....758
ITenantGroup interface....761
ITenantGroup properties....762
ITenantGroup methods....763
ITenantGroup examples....764
IUser interface....766
IUser properties....766
IUser methods....767
IUser examples....767
IUserFieldPermission interface....768
IUserFieldPermission properties....768
IUserFieldPermission methods....768
IUserPermission interface....769
IUserPermission properties....769
IUserPermission methods....769
IUserTablePermission interface....770
IUserTablePermission properties....770
IUserTablePermission methods....771

Appendix G: Database Administration Entity Collection Interface Reference....773

Collection interface naming conventions....774
Common collection features....775
IAreaSet interface....775
IAreaSet properties....776
IAreaSet methods....776
IAuthenticationSystemSet interface....776
IAuthenticationSystemSet properties....776
IAuthenticationSystemSet methods....777
IDataAdminCollection interface....777
IDataAdminCollection properties....777
IDataAdminCollection methods....778
IDomainSet interface....780
IDomainSet properties....780
IDomainSet methods....780
IExtentList interface....781
IExtentList properties....781
IExtentList methods....781
IFieldPermissionSet interface....782
IFieldPermissionSet properties....782
IFieldPermissionSet methods....782
IFieldSet interface....782

IFieldSet properties....783
IFieldSet methods....783
IIndexSet interface....783
IIndexSet properties....783
IIndexSet methods....783
IIterator interface....784
IIterator methods....784
Examples for using the Iterator method....785
IPartitionCollection interface....785
IPartitionCollection properties....785
IPartitionCollection methods....786
IPartitionMap interface....787
IPartitionMap properties....787
IPartitionMap methods....787
IPartitionPolicyDetailSet interface....789
IPartitionPolicyDetailSet properties....789
IPartitionPolicyDetailSet methods....789
IPartitionPolicySet interface....790
IPartionPolicySet properties....791
IPartionPolicySet methods....791
ISchemaPartitionMap interface....791
ISchemaPartitionMap properties....791
ISchemaPartitionMap methods....791
ISequenceSet interface....793
ISequenceSet properties....793
ISequenceSet methods....793
ISequenceValueMap interface....794
ISequenceValueMap properties....794
ISequenceValueMap methods....794
ITablePermissionSet interface....795
ITablePermissionSet properties....795
ITablePermissionSet methods....795
ITableSet interface....796
ITableSet properties....796
ITableSet methods....796
ITenantGroupSet interface....797
ITenantGroupSet properties....797
ITenantGroupSet methods....797
ITenantSet interface....798
ITenantSet properties....798
ITenantSet methods....798
IUserFieldPermissionSet interface....799
IUserFieldPermissionSet properties....800
IUserFieldPermissionSet methods....800
IUserPermissionSet interface....800

IUserPermissionSet properties....800
IUserPermissionSet methods....800
IUserSet interface....801
IUserSet properties....801
IUserSet methods....801
IUserTablePermissionSet interface....802
IUserTablePermissionSet properties....802
IUserTablePermissionSet methods....803

Index....805

Preface

Purpose

Many operating systems and user interfaces provide tools that allow one application to exchange data or use services provided by another application (external program). OpenEdge allows you to use some of these tools as external program interfaces (EPIs) from within an OpenEdge application.

This is a three-part book that discusses topics pertaining to external program interfaces. The first part of this book describes information you need to know about accessing databases. The second part discusses input and output processes. Finally, the third part of this manual describes the EPIs that OpenEdge supports, and explains how to use them from ABL (Advanced Business Language) to integrate your OpenEdge application with other applications in your operating environment.

Audience

This manual is intended for any ABL programmer who is writing applications that require the use of external programming interfaces. In general, this programmer has a working knowledge of both the EPIs and the operating systems in which the EPIs run.

Organization

Part 1, Data Management

Database Access on page 35

Describes ABL statements that allow you to read and write data in a database. These topics also describe techniques for browsing database data on a display.

Application Security on page 97

Surveys the types of data security that ABL supports. These topics focus specifically on how you can provide various types of run-time security within an application.

Multi-tenant ABL on page 189

Describes how ABL executes in a multi-tenant environment and the ABL syntax available for a super-tenant user to create and access database data for one or more regular tenants.

ABL API for Multi-tenant and Table Partition Management on page 205

Describes how to maintain a multi-tenant enabled database using the ABL API for Multi-tenant Maintenance, which includes OpenEdge-provided, ABL user-defined classes and interfaces that allow you to perform Data Dictionary operations to modify the database multi-tenant schema.

Auditing on page 241

Surveys auditing support in OpenEdge applications. OpenEdge provides core support for recording secure audit trails in any OpenEdge RDBMS that is enabled for auditing.

Part 2, Input/Output Processes

Handling User Input on page 263

Describes how OpenEdge handles input from the keyboard and the mouse. These topics include examples of how you can monitor data entry from the user.

Alternate I/O Sources on page 297

Describes how your OpenEdge application can handle I/O to and from operating system files and special devices.

Colors and Fonts on page 325

Surveys OpenEdge support for colors and fonts, including the techniques for modifying colors and fonts within an application.

Creating Reports on page 343

Describes how to generate and format simple and complex reports.

Part 3, External Program Interfaces

Introduction to External Program Interfaces on page 413

Describes the EPIs supported by OpenEdge, what they can provide for your applications, and the requirements for using them.

System Clipboard on page 437

Describes how to use the CLIPBOARD handle to read and write data to the system clipboard, and how to provide user interactions between the system clipboard and your OpenEdge application.

Providing Help for OpenEdge Applications on page 449

Describes how to provide online help for your OpenEdge application.

Named Pipes on page 465

Describes how to use named pipes to provide interprocess communication (IPC) between your OpenEdge application and another application running on either the UNIX operating system or Windows. It emphasizes techniques that enable any OpenEdge application to be a data server for the external application.

Shared Library and DLL Support on page 483

Describes how to call UNIX shared library functions and Windows dynamic link library (DLL) functions from a OpenEdge application. This includes how to declare shared library functions as internal procedures and how to pass OpenEdge data items as shared library parameters.

Windows Dynamic Data Exchange on page 507

Describes how to use dynamic data exchange (DDE) in Windows to send and receive data between your OpenEdge application running as a DDE client and another application running as a DDE server. Note that DDE is a deprecated feature supported only for backward compatibility. Consider using COM instead.

Using COM Objects in ABL on page 523

Describes OpenEdge support for COM objects, including information common to both ActiveX Automation objects and ActiveX Controls.

ActiveX Automation Support on page 543

Describes OpenEdge support for ActiveX Automation and how to implement an OpenEdge application as an ActiveX Automation Controller from ABL.

ActiveX Control Support on page 551

Describes OpenEdge support for ActiveX controls in ABL, including how to convert an earlier application using VBX controls to the same application using ActiveX controls.

Sockets on page 579

Describes OpenEdge support for the use of sockets in ABL, including connecting to and disconnecting from a port using sockets and receiving and transmitting data.

Host Language Call Interface on page 375

Describes the OpenEdge Host Language Call (HLC) Interface.

Appendices

COM Object Data Type Mapping on page 609

Describes the automatic conversion support between COM data types and ABL data types for COM object properties, methods, and events.

Audit Policy Maintenance APIs on page 617

Describes several ABL APIs used to implement the OpenEdge Audit Policy Maintenance tool. This tool allows you to create and maintain audit policies in an OpenEdge RDBMS.

HLC Library Function Reference on page 671

Describes the HLC library functions that provide an interface between your HLC functions and ABL.

Command and Utility Reference on page 701

Describes the Quoter utility.

Database Administration Entity Interface Reference on page 707

Describes the interface objects available with the ABL API for Multi-tenant Maintenance that allow you to access and update the individual entities of a multi-tenant schema.

Database Administration Entity Collection Interface Reference on page 773

Describes the collection interface objects available with the ABL API for Multi-tenant Maintenance that allow you to access and navigate through multiple entity interface objects at one time.

Documentation conventions

See Documentation Conventions for an explanation of the terminology, format, and typographical conventions used throughout the OpenEdge content library.

Purpose

Many operating systems and user interfaces provide tools that allow one application to exchange data or use services provided by another application (external program). OpenEdge allows you to use some of these tools as external program interfaces (EPIs) from within an OpenEdge application.

This is a three-part book that discusses topics pertaining to external program interfaces. The first part of this book describes information you need to know about accessing databases. The second part discusses input and output processes. Finally, the third part of this manual describes the EPIs that OpenEdge supports, and explains how to use them from ABL (Advanced Business Language) to integrate your OpenEdge application with other applications in your operating environment.

Audience

This manual is intended for any ABL programmer who is writing applications that require the use of external programming interfaces. In general, this programmer has a working knowledge of both the EPIs and the operating systems in which the EPIs run.

Organization

Part 1, Data Management

Database Access on page 35

Describes ABL statements that allow you to read and write data in a database. These topics also describe techniques for browsing database data on a display.

Application Security on page 97

Surveys the types of data security that ABL supports. These topics focus specifically on how you can provide various types of run-time security within an application.

Multi-tenant ABL on page 189

Describes how ABL executes in a multi-tenant environment and the ABL syntax available for a super-tenant user to create and access database data for one or more regular tenants.

ABL API for Multi-tenant and Table Partition Management on page 205

Describes how to maintain a multi-tenant enabled database using the ABL API for Multi-tenant Maintenance, which includes OpenEdge-provided, ABL user-defined classes and interfaces that allow you to perform Data Dictionary operations to modify the database multi-tenant schema.

Auditing on page 241

Surveys auditing support in OpenEdge applications. OpenEdge provides core support for recording secure audit trails in any OpenEdge RDBMS that is enabled for auditing.

Part 2, Input/Output Processes

Handling User Input on page 263

Describes how OpenEdge handles input from the keyboard and the mouse. These topics include examples of how you can monitor data entry from the user.

Alternate I/O Sources on page 297

Describes how your OpenEdge application can handle I/O to and from operating system files and special devices.

Colors and Fonts on page 325

Surveys OpenEdge support for colors and fonts, including the techniques for modifying colors and fonts within an application.

Creating Reports on page 343

Describes how to generate and format simple and complex reports.

Part 3, External Program Interfaces

Introduction to External Program Interfaces on page 413

Describes the EPIs supported by OpenEdge, what they can provide for your applications, and the requirements for using them.

System Clipboard on page 437

Describes how to use the CLIPBOARD handle to read and write data to the system clipboard, and how to provide user interactions between the system clipboard and your OpenEdge application.

Providing Help for OpenEdge Applications on page 449

Describes how to provide online help for your OpenEdge application.

Named Pipes on page 465

Describes how to use named pipes to provide interprocess communication (IPC) between your OpenEdge application and another application running on either the UNIX operating system or Windows. It emphasizes techniques that enable any OpenEdge application to be a data server for the external application.

Shared Library and DLL Support on page 483

Describes how to call UNIX shared library functions and Windows dynamic link library (DLL) functions from a OpenEdge application. This includes how to declare shared library functions as internal procedures and how to pass OpenEdge data items as shared library parameters.

Windows Dynamic Data Exchange on page 507

Describes how to use dynamic data exchange (DDE) in Windows to send and receive data between your OpenEdge application running as a DDE client and another application running as a DDE server. Note that DDE is a deprecated feature supported only for backward compatibility. Consider using COM instead.

Using COM Objects in ABL on page 523

Describes OpenEdge support for COM objects, including information common to both ActiveX Automation objects and ActiveX Controls.

ActiveX Automation Support on page 543

Describes OpenEdge support for ActiveX Automation and how to implement an OpenEdge application as an ActiveX Automation Controller from ABL.

ActiveX Control Support on page 551

Describes OpenEdge support for ActiveX controls in ABL, including how to convert an earlier application using VBX controls to the same application using ActiveX controls.

Sockets on page 579

Describes OpenEdge support for the use of sockets in ABL, including connecting to and disconnecting from a port using sockets and receiving and transmitting data.

Host Language Call Interface on page 375

Describes the OpenEdge Host Language Call (HLC) Interface.

Appendices

COM Object Data Type Mapping on page 609

Describes the automatic conversion support between COM data types and ABL data types for COM object properties, methods, and events.

Audit Policy Maintenance APIs on page 617

Describes several ABL APIs used to implement the OpenEdge Audit Policy Maintenance tool. This tool allows you to create and maintain audit policies in an OpenEdge RDBMS.

HLC Library Function Reference on page 671

Describes the HLC library functions that provide an interface between your HLC functions and ABL.

Command and Utility Reference on page 701

Describes the Quoter utility.

Database Administration Entity Interface Reference on page 707

Describes the interface objects available with the ABL API for Multi-tenant Maintenance that allow you to access and update the individual entities of a multi-tenant schema.

Database Administration Entity Collection Interface Reference on page 773

Describes the collection interface objects available with the ABL API for Multi-tenant Maintenance that allow you to access and navigate through multiple entity interface objects at one time.

Documentation conventions

See Documentation Conventions for an explanation of the terminology, format, and typographical conventions used throughout the OpenEdge content library.

Data Management

For details, see the following topics:

- Database Access
- Application Security
- Multi-tenant ABL
- ABL API for Multi-tenant and Table Partition Management
- Auditing

1

Database Access

These topics describe how to access records in an OpenEdge database.

For details, see the following topics:

- Database connections
- Disconnecting databases
- Logical database names
- Database aliases
- Data-handling statements
- Adding and deleting records
- Defining a set of records to fetch
- Fetching records
- Fetching field lists
- Joining tables
- The CONTAINS operator
- Sequences
- Database trigger considerations
- Using the RAW data type
- Multi-database programming techniques

- Creating schema cache files

Database connections

An OpenEdge application can access one or more OpenEdge or non-OpenEdge databases simultaneously. The databases can be located on different operating systems using different networking protocols. To access non-OpenEdge databases, you must use the appropriate DataServer, such as DataServer for Oracle. You must connect to a database before you can access it. There are four ways to connect to a database:

- As a command-line argument (connection parameter) when starting OpenEdge
- With the `CONNECT` statement (in the Procedure Editor or in an ABL—Advanced Business Language—procedure or class)
- With the OpenEdge Data Dictionary or other database administration tools
- Using the auto-connect feature

Note that a multi-user application can simultaneously connect to a database for only as many times as specified in the Number of Users (`–n`) startup parameter. For information on exceeding this user count, see Connection failures and disruptions on page 40. For more information on database connection management, see *Startup Command and Parameter Reference* and *Manage ABL Applications*.

Connection parameters

All of the database connection techniques let you use connection parameters. For more information on these parameters, see *Startup Command and Parameter Reference*.

For multiple database programming, the most important connection parameter is the Physical Database Name (`–db`) startup parameter, which allows you to specify multiple databases.

The Number of Databases (`–h`) startup parameter lets you set the number of connected databases allowed during a OpenEdge session, up to a maximum of 240 (the default is 5).

You can group connection parameters in a text file called a *parameter file*. The Parameter File (`–pf`) connection parameter invokes the parameter file when you connect to a database. Parameter files ordinarily have `.pf` file extensions. For more information on parameter files, see *Startup Command and Parameter Reference*.

You can also create secure database connections using the TLS-based Connections (`-ssl`) startup parameter. For more information, see Secure database connections on page 36.

Secure database connections

You can use the Transport Layer Security (TLS) to provide a security infrastructure that protects communications between a database client and server. TLS provides data privacy over network connections and authentication between clients and servers on those connections using elements of Public Key Infrastructure (PKI). These elements include private and public keys that the clients and servers use to authenticate each other and to set up data encryption and decryption services between the initiator of the communications (TLS client) and the responder (TLS server).

The server is identified by the private key that it stores and the client is identified as a valid TLS client for that server by the public key that it stores and provides to the server. TLS clients gain access to public keys using digital (public key) certificates provided by a trusted certificate authority (CA) that also provides the private key confidentially to the TLS server.

Note: TLS incurs heavy performance penalties, depending on the client, server, and network resources and load.

To secure a connection to the database using TLS:

1. Ensure that your OpenEdge RDBMS is TLS-enabled. This requires that you:
   a. Install a private key and server certificate on the database server or use the default private key and certificate installed with the server by OpenEdge.
   b. Startup the database server with the startup parameters required to enable TLS database connections.
2. Ensure that your ABL client system has a public key certificate installed that corresponds to the private key installed and used to support TLS on the database server.
3. Using the startup command line or the `CONNECT( )` statement, connect to the database using the client connection parameters required to establish a TLS connection, as shown in the following table.

Table 1: TLS database connection parameters

Connection parameter	Description
TLS-based Connection (`-ssl`)	Specifies a Transport Layer Security (TLS) connection. (Required)
No Host Verify (`-nohostverify`)	Turns off host verification for a Transport Layer Security (TLS) connection. (Optional)
No Session Reuse (`-nosessionreuse`)	Avoids the reuse of a Transport Layer Security (TLS) session ID. (Optional)

For more information on:

- OpenEdge TLS support and how to manage private and public keys and certificates to secure connections, see *Learn about Security and Auditing*
- How to enable a database server for TLS connections, see *Manage the OpenEdge Database*
- The database server startup and connection parameters used to enable TLS, see *Startup Command and Parameter Reference*

The CONNECT statement

The `CONNECT` statement allows you to connect to a database from an ABL procedure or directly from the Procedure Editor. The `CONNECT` statement has the following syntax:

Syntax

```
CONNECT { physical-name      [ parameters ]
        | -db physical-name  [ parameters ]
        | -pf parameter-file [ parameters ]
      }
      [
        { -db physical-name  [ parameters ]
        | -pf parameter-file [ parameters ]
        }
      ]...[ NO-ERROR ]
```

physical-name

An argument that represents the actual name of the database on a disk. The first *physical–name* argument you specify in a CONNECT statement does not require the Database Name (–db) parameter. All subsequent *physical–names* must be preceded by the –db parameter.

parameter-file

The name of a parameter file that contains database connection information. See *Startup Command and Parameter Reference* for more information on parameter files.

parameters

One or more database connection parameters. Each connection parameter applies to the most recently specified database (–db). For the parameters you can specify, see the information on client connection parameters in the *Startup Command and Parameter Reference*.

NO-ERROR

This argument suppresses the error condition, but still displays the error message when an attempt to CONNECT to a database fails.

Although you can connect to several databases within one CONNECT statement, it is a good idea to connect only one database per CONNECT statement, because a connection failure for one database causes a termination of the current CONNECT statement. However, databases already connected when the statement terminates stay connected. In cases like this, it is a good idea to use the CONNECTED functions to see which databases were connected and which were not.

Here is an example of using a parameter file with the CONNECT statement:

```
CONNECT -pf parm3.pf.
```

In this example, the CONNECT statement uses the parm3.pf file to connect to the appldb1 database.

parm3.pf

```
-db appldb1
```

A single procedure cannot connect to and then access a database. The following code fragment does not run:

```
/* NOTE: this code does NOT work */

CONNECT sports2000.
FOR EACH sports2000.Customer NO-LOCK:
  DISPLAY Customer.
END.
```

By splitting this example into a procedure and a subprocedure, you can connect to the database in the main procedure, and then access it in the subprocedure. For example:

i-topproc.p

```
CONNECT sports2000.
RUN i-subproc.p.
```

i-subproc.p

```
FOR EACH sports2000.Customer NO-LOCK:
  DISPLAY Customer.
END.
```

For more information on the `CONNECT` statement, see *ABL Reference*.

Auto-connect

The auto-connect feature uses information stored in one database to connect to a second database at runtime. The database that contains the connect information is the primary application database; it must be connected before OpenEdge can execute the auto-connect.

OpenEdge executes the auto-connect when a precompiled procedure references data from the second database. It executes immediately prior to running the precompiled procedure. It does not work with procedures run from the Editor or otherwise compiled on-the-fly.

If you use a `CONNECT` statement while you are connected to the primary database, OpenEdge merges the information in the `CONNECT` statement with the information in the primary database's auto-connect list. If there is a conflict, the information in the `CONNECT` statement takes precedence. For more information, see the `CONNECT` Statement reference entry in *ABL Reference*.

For information on how to set up an auto-connect list in the primary database, see *Manage ABL Applications*.

Multi-database connection considerations

When you develop a multi-database application, keep in mind the following information on database connections:

- Connect all databases accessed by the application at compile time. Once compiled, the application can run no matter where you store the databases or how you connect to them.
- Establish unique logical names for each of the databases you connect. Use these names to reference the databases within your application. These names are stored in the r-code of the application at compilation time. For more information on logical database names, see Logical database names on page 43.

Run-time connection considerations

As mentioned in Database connections on page 36, there are various ways to connect at runtime. For multi-database applications, you must decide which technique to use.

When connecting in the run-time environment, you must use the same logical database names that you used during compilation. If you need to use a different logical database name for some reason, you may add an alias to the database to allow r-code files with other names to run.

Connection failures and disruptions

Database connections might fail for a number of reasons, such as:

- The database server for multi-user database access is not available—perhaps the network is down, or the network has not been started, or the network parameters are incorrect.
- The logical name of the application database already exists for the current session.
- The syntax for a connection parameter is wrong.
- The connection exceeds the maximum number of users per database (–n).

The following table lists the default failure behavior for each of the run-time connection techniques.

Table 2: Connection failure behavior

Connection technique	Default connection failure behavior
At OpenEdge startup	The OpenEdge session does not run.
`CONNECT` statement	The procedure executes up to the `CONNECT` statement where the connection failure occurs. The connection failure raises the error condition for the procedure. OpenEdge error processing does not undo database connections or disconnections. Any database connected in the procedure before the failed database connection remains connected.
Auto-connect	The procedures that contain a reference to the auto-connect database do not run, and a stop or break condition results in the calling procedure.

OpenEdge displays any connection error messages at the point of failure.

Before running a procedure or subprocedure, OpenEdge checks the procedure for database references. For each reference, OpenEdge searches through all of the connected databases, trying to find the corresponding database. If the database is not found, OpenEdge again searches all of the connected databases for an auto-connect list with an entry for that database. If an entry is found, OpenEdge connects the database. If no entry is found, OpenEdge does not run the procedure and raises a stop or break condition in the calling procedure.

Server, network, or machine failures can disrupt an application, even after a successful connection. If a failure occurs while a subprocedure is accessing a database, OpenEdge raises a stop or break condition in the calling procedure.

The following sections present a number of recommendations to help you manage database connection failures and disruptions in an application.

Using CONNECT with NO-ERROR

If you designate a startup procedure with the Startup Procedure (–p) parameter, always use the NO–ERROR option with the CONNECT statement. If a CONNECT statement fails, the NO–ERROR option suppresses the resulting error condition, allowing the procedure to continue executing. For more information on using the NO–ERROR option, see *ABL Error Handling*.

After a connection failure, if a subprocedure tries to access the unconnected database, OpenEdge raises a STOP (or break) condition in the calling procedure. Therefore, before you execute any subprocedures that access a database, test whether the database is connected. You can do this with the CONNECTED function.

Using the CONNECTED function

The CONNECTED function tests whether a database is connected. This function helps you to route program control around portions of an application that might be affected by database connection failures and disruptions. This example tests a database connection with the CONNECTED function:

```
IF NOT CONNECTED(logical-name) THEN DO:
  MESSAGE "CONNECTING logical-name".
  CONNECT logical-name ... NO-ERROR.
END.
IF CONNECTED(logical-name) THEN
  RUN procedure.
ELSE
  MESSAGE "DATABASE NOT AVAILABLE".
```

If the database is not connected, the above code attempts to connect the database. This example only runs the procedure if the database is connected. Auto-connect precludes the use of the CONNECTED function to test for database connections. For more information on the CONNECTED function, see *ABL Reference*.

OpenEdge database connection functions

The following table lists the ABL functions that allow you to test database connections, get information on connected databases, and get information on the types of databases that you can access.

Table 3: ABL database functions

ABL function	Description
CONNECTED	Tests whether a given database is connected
DATASERVERS	Returns a character string containing a list of database types supported by the installed OpenEdge product; for example, "OpenEdge,ORACLE."
DB-REMOTE-HOST	Returns a character string containing the IP address of the database connection
DBRESTRICTIONS	Returns a character string that describes the OpenEdge features that are not supported for a particular database, for example, if the database is an ORACLE database, the return string is: "LAST,PREV,RECID,SETUSERID"

ABL function	Description
DBTYPE	Returns the database type of a currently connected database, for example "OpenEdge," "ORACLE," etc.
DBVERSION	Returns the version of the currently connected database
FRAME–DB	Returns a character string that contains the logical name of the database for the field in which the cursor was last positioned for input
NUM–DBS	Returns the number of connected databases
LDBNAME	Returns the logical name of a currently connected database
PDBNAME	Returns the physical name of a currently connected database
SDBNAME	Returns the logical name of a schema holder for a database

For more information on these functions, see *ABL Reference*.

Use these functions to perform various tasks related to connection, such as determining connection status. The `i-infor.p` procedure displays a status report for all connected databases.

i-infor.p

```
DEFINE VARIABLE ix AS INTEGER NO-UNDO.

DO ix = 1 TO NUM-DBS WITH DOWN:
  DISPLAY PDBNAME(ix) LABEL "Physical Database"
    LDBNAME(ix) LABEL "Logical Name"
    DBTYPE(ix) LABEL "Database Type"
    DBRESTRICTIONS(ix) LABEL "Restrictions"
    SDBNAME(LDBNAME(ix)) LABEL "Schema Holder DB".
END.
```

Balancing conserving connections with minimizing overhead

As more and more users connect to a database, the number of available connections decreases. You can conserve the number of connections by connecting temporarily, and then disconnecting when the application is done accessing the database. This frees up the connection for another user. If the number of available connections is scarce, an application should release connections wherever possible.

However, each connect and disconnect generates *connection overhead*, which is the sum of operations necessary for an application to connect and disconnect a database. The time used for connection overhead depends on the nature of the database connections. A connection to a database over a network generally takes longer than a connection to a local database. While the application connects to a database, the end user waits.

When there are plenty of available connections, you might want to reduce the number of connects and disconnects to minimize overhead. The best way to do this is to connect all application databases only once at application startup. With this technique, all connection overhead occurs once at application startup and does not occur again throughout the life of the application.

Disconnecting databases

By default, OpenEdge disconnects all databases at the end of a session. You can explicitly disconnect a database with the DISCONNECT statement, which has the following syntax:

Syntax

```
DISCONNECT logical-database-name
```

The *logical–database–name* represents the logical name of a connected database. It can be an unquoted string, a quoted string, or a character expression.

A DISCONNECT statement does not execute until all active procedures that reference the database end or stop. In the following procedures, the mydb database is not disconnected until the end of the i-subproc1.p procedure:

i-mainprc1.p

```
CONNECT -db mydb -1.
RUN i-subproc1.p.
```

i-subproc1.p

```
RUN i-subproc2.p.

FOR EACH mydb.Customer
  EXCLUSIVE-LOCK:
  UPDATE Customer.Name Customer.Address Customer.City Customer.State
    Customer.PostalCode.
END.
```

i-subproc2.p

```
DISCONNECT mydb.
```

Logical database names

When you connect to a database, OpenEdge automatically assigns that database a default logical name for the current OpenEdge session. The default logical name consists of the physical database name without the .db file extension. For example, if you connect to a database with the physical name mydb1.db, the default logical database name is mydb1. OpenEdge uses the logical database name mydb1 to resolve database references, and stores it in the compiled r-code of any procedures that you compile that reference the mydb1.db database.

The Logical Database Name (–ld) connection parameter allows you to specify a logical database name other than the default.

The following example establishes the logical name `firstdb` for the physical database `mydb1.db` during the current OpenEdge session:

```
pro mydb1 -ld firstdb
```

When you develop and compile an application to run on the `mydb1.db` database, it is the logical name, not the physical name, that OpenEdge stores in the r-code. You must use the logical name `firstdb` in your procedures (`.p`) to reference the `mydb1.db` database.

Logical database names allow you to change physical databases without recompiling an application. To run a compiled application on a new physical database without recompiling, the new database must have identical structure and time stamp or Cyclic Redundancy Check (CRC) values for the tables accessed by the application and must be connected with the same logical name (or alias) used to compile the application. For example:

```
pro mydb2 -ld firstdb
```

The previous example establishes the logical name `firstdb` for a new physical database `mydb2.db`.

Note: OpenEdge does not allow you to run the OpenEdge Data Administration tool or character Data Dictionary against a database connected with the logical name `DICTDB`.

A database connection fails if the logical database name of the database that you connect to has the same logical name as an already connected database of the same database type (OpenEdge, ORACLE, etc.). If you try to do this, OpenEdge assumes that database is already connected and ignores the request.

For information about the characters allowed in the logical name, see *Startup Command and Parameter Reference*.

Database aliases

An *alias* is a synonym for a logical database name. You use an alias as a database reference in ABL procedures in place of a logical database name.

You establish a logical database name when you connect a OpenEdge session to a physical database. You create and assign an alias to a logical database name of an already connected database using the `CREATE ALIAS` statement. By reassigning an alias to different logical database names, you can run a compiled procedure on other connected databases that have identical structure and time stamps or CRC values for the tables referenced by the procedure. A logical database name can have more than one alias, but each alias refers to only one logical database name at a time.

The OpenEdge Data Dictionary offers an example of alias usage. The OpenEdge Data Dictionary is a general-purpose OpenEdge application that works on any database that has `DICTDB` as an alias. The first database connected during a OpenEdge session automatically receives the alias `DICTDB`. During a OpenEdge session, you can reassign the `DICTDB` alias to another connected database with the **Select Working Databases** option on the **Database** menu in the OpenEdge Data Dictionary.

Note: Because of the need to reassign the `DICTDB` alias, OpenEdge does not allow you to run the OpenEdge Data Dictionary against a database connected with the logical name `DICTDB`.

Use the CREATE ALIAS statement during a OpenEdge session to assign or reassign an alias to a connected database. You can use this statement from the ABL editor or from an application procedure. This is the syntax for the CREATE ALIAS statement:

Syntax

```
CREATE ALIAS alias FOR DATABASE logical-name [ NO-ERROR ]
```

The *alias* argument can be an unquoted string, a quoted string, or an expression. The *logical–name* argument represents the existing logical name of a connected database. It can be an unquoted string, a quoted string, or an expression. You cannot create an alias that is the same as the logical database name of a connected database. The named database must be connected unless you use the NO–ERROR option.

When you create an alias, OpenEdge logs the alias assignment in a table that remains in memory for the current session. If you use the DISCONNECT statement to disconnect a database from within an application, all existing aliases assigned to the logical database name remain in the alias table until the end of the OpenEdge session. Later, if you connect to a database with the same logical database name during the same OpenEdge session, you can use the same aliases to reference that logical database name. If you create an alias that already exists in the session alias table, OpenEdge replaces the existing alias with the new alias. This allows you to reassign existing aliases to new logical database names.

The DELETE ALIAS statement allows you to delete an alias from the alias table of the current OpenEdge session. The following is the syntax for the DELETE ALIAS statement:

Syntax

```
DELETE ALIAS alias
```

The *alias* argument represents an alias that exists in the current alias session table.

Creating aliases in applications

You cannot assign and reference an alias in the same procedure. You must assign an alias to a logical database name prior to compiling and running procedures that use that alias. For example, the procedure in i-alias1.p fails to compile when it reaches the FOR EACH statement, because you cannot assign and reference the alias myalias in a single procedure.

i-alias1.p

```
/* Note that this procedure does not work */
CREATE ALIAS myalias FOR DATABASE sports2000.

FOR EACH myalias.Customer NO-LOCK:
  DISPLAY Customer.
END.
```

To solve this problem, split i-alias1.p into two procedures, as in the following examples:

i-alias2.p

```
CREATE ALIAS myalias FOR DATABASE sports2000.
RUN i-dispcust.p.
```

i-dispcust.p

```
FOR EACH myalias.Customer NO-LOCK:   /* myalias.Customer */
  DISPLAY Customer.                  /* myalias.Customer */
END.
```

Compiling procedures with aliases

As a general rule, you should not compile procedures while using aliases. This potentially leads to confusion about what database name ends up in the r-code. For example, the OpenEdge dictionary programs which contain `DICTDB` as a qualifier must be compiled in a session where the database concerned has the logical name `DICTDB`. That way, when the dictionary r-code is run, it can run for any database name as long as the database uses the `DICTDB` alias. In summary, logical names are useful for compiling, aliases are useful at runtime.

However, you can still compile procedures using aliases. Suppose you have three databases called `eastdb`, `centraldb`, and `westdb`, all of which contain `Customer` tables with identical structure and time stamps or CRC values. Your application requires a general report procedure that can run against any of these `Customer` tables. To begin developing your general report procedure, start OpenEdge and connect to the `eastdb`, `centraldb`, or `westdb` database using the logical name `myalias`. Develop and compile the customer report procedure using `myalias` to prefix table and field references as shown in the previous procedure `i-dispcust.p`. All unqualified table and field references in the report procedure `i-dispcust.p` resolve to the `myalias` logical name at compilation time. When you finish compiling your procedure, disconnect from the database represented by the `myalias` logical database name.

You must assign an alias to the logical database name of a connected database prior to running any procedure that uses that alias as a database reference. Therefore, you need to develop a procedure that uses the `CREATE ALIAS` statement to assign the `myalias` alias to a logical database name and run the report procedure (`i-dispcust.p`) as a subprocedure. See the `i-alias2.p` procedure above.

Although it is not recommended because of the possible confusion it can cause, you may need to compile a procedure with an alias. To do this, you must know how OpenEdge places database references in the r-code of the procedure at compilation time. Remember, you must assign the alias to a logical database name prior to compiling the procedure. In general, use only the alias as a database prefix for all table and field references in the general-purpose procedures, and always fully qualify every database reference within such a procedure. Use the following syntax:

Syntax

```
alias.table-name alias.table-name.field-name
```

If you use this syntax for every table or field reference in your procedure, only the alias will be represented as the database reference in the procedure's r-code after compilation. Note that this is the only exception to the rule that you should never compile using aliases.

Unqualified table and field references within procedures may cause both the alias and the logical database name for a particular physical database to be represented in the r-code for the procedure at compilation time.

Using shared record buffers with aliases

Be careful when using shared buffers with aliases. If you reference a shared buffer after changing the alias that initially was used in defining it, a run-time error results. For example, consider the following procedures:

i-main2.p

```
CREATE ALIAS myalias FOR DATABASE sports1.
RUN i-makebuf.p.
```

i-makebuf.p

```
DEFINE NEW SHARED BUFFER mybuf FOR myalias.customer.

CREATE ALIAS myalias FOR DATABASE sports2.
RUN i-disp.p
```

i-disp.p

```
DEFINE SHARED BUFFER mybuf FOR myalias.Customer.

FOR EACH mybuf:
  DISPLAY mybuf.
END.
```

In this example, procedure `i-main2.p` calls `i-makebuf.p`, which in turn calls `i-disp.p`. The alias `myalias` is created in `main.p`, with reference to database `sports1`. In `i-makebuf.p`, the shared buffer `mybuf` is defined for the table myalias.Customer. Then, in the next line, `myalias` is changed, so that it now refers to database `sports2`. When an attempt is made to reference shared buffer `mybuf` in procedure `i-disp.p`, a run-time error occurs, with the message "i-disp.p unable to find shared buffer for mybuf."

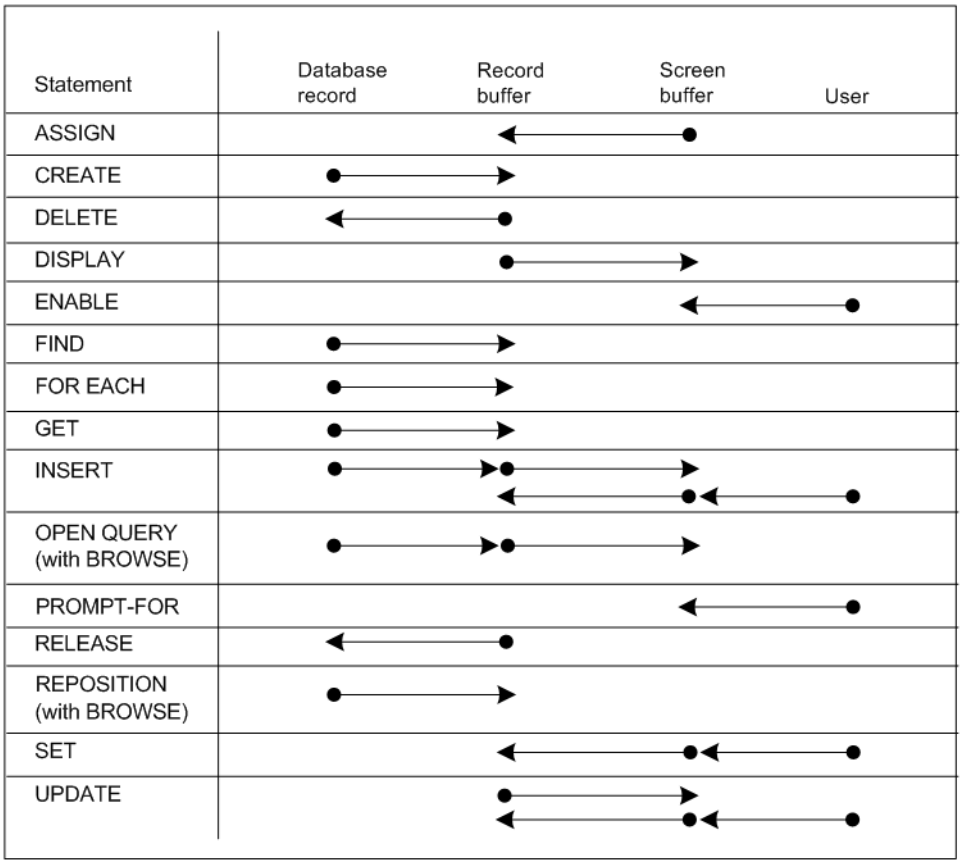
Data-handling statements

Statements that move data from one location to another are called *data-handling statements*.

ABL stores data in various locations—a database, a record buffer, a screen buffer, etc. A database stores application data on disk. A record buffer stores data temporarily while a procedure accesses the data, and stores the values of variables used in the procedure. A screen buffer stores data being displayed on the screen or being sent to another output destination; it also stores data that is being entered from the terminal or other input source.

The following figure shows how the data-handling statements move data.

Figure 1: Data movement



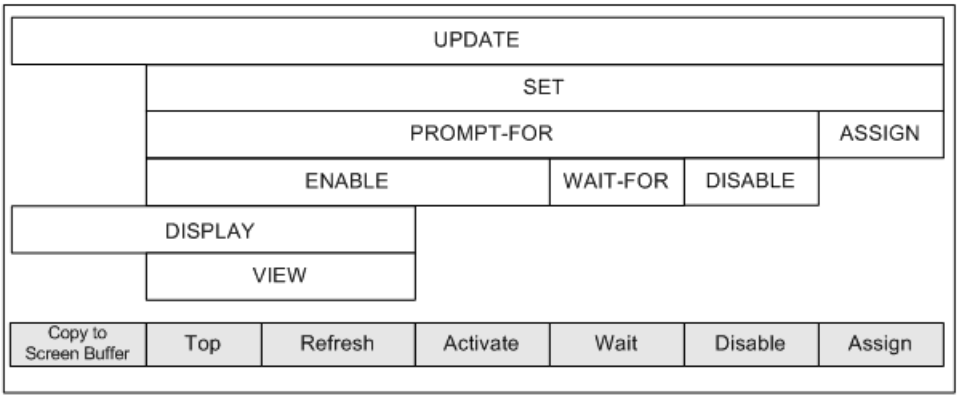
To use the above figure, read each statement from top to bottom. The beginning of the first arrow (the dot) indicates the source of the data. The arrowhead of the last arrow indicates where the data is finally stored. For example, the `DISPLAY` statement gets data from a record buffer and moves it into a screen buffer. Once in the screen buffer, the data is displayed on screen. Also, note that `INSERT` creates a new database record, moves it to the record buffer, and then to the screen buffer where the user enters data, which is then moved back to the record buffer.

Some statements are made up of other statements. For example, the `UPDATE` statement is made up of the `DISPLAY`, `PROMPT-FOR`, and `ASSIGN` statements, and performs the same steps as these other statements. First, it copies data from the record buffer to the screen buffer (`DISPLAY`). Second, it allows data to be entered into the screen buffer (`PROMPT-FOR`). Third, it copies the data back to the record buffer (`ASSIGN`).

You can use statements as building blocks, using only as many as you need to do specific tasks. For example, the `INSERT` statement is very powerful and combines several processing steps into one statement. But in some situations, it is less flexible than using the `CREATE`, `DISPLAY`, and `SET` statements individually (or the `CREATE` and `UPDATE` statements).

The following figure shows the primary data-handling statements and shows which statements are composed of other statements.

Figure 2: The primary data-handling statements



Note that the UPDATE, SET, and ASSIGN statements do not actually write records to the database. However, at the end of a transaction (or at the end of the record scope or after an explicit RELEASE), ABL writes all modified records to the database.

If you modify a record using INSERT, UPDATE, or SET, the ABL Virtual Machine (AVM) assigns the change to the record buffer (and, hence, eventually to the database). However, if you use the ENABLE statement, you must explicitly assign any changes with the ASSIGN statement.

For more information on the statements in the above figure, see *ABL Reference*.

Adding and deleting records

To add records to a database, you can use either the CREATE or INSERT statements. The CREATE statement places a newly created record in the database, but does not display the record or request user input. All fields in the record are set to the initial values specified in the Data Dictionary. The CREATE statement causes any CREATE trigger associated with the table to execute. This trigger may set fields in the record to new values. The INSERT statement not only creates the record, but also displays the record and requests input.

The INSERT statement is composed of four other statements—the CREATE, DISPLAY, PROMPT–FOR and ASSIGN statements. Three of these statements (DISPLAY, PROMPT–FOR, and ASSIGN) comprise the UPDATE statement. You can emulate an INSERT statement by using the four statements, or by using the CREATE and UPDATE statements. This is more flexible than using the INSERT statement alone.

For example, the INSERT statement always displays fields in the order they are defined in the schema. The UPDATE statement lets you specify the order they are displayed. Note that if you use an INSERT statement, the CREATE trigger is executed before the record is displayed.

To delete records from a database, use the DELETE statement. The DELETE statement causes any DELETE trigger associated with the table to execute.

Defining a set of records to fetch

To fetch records, you must first define the set of records that you want ABL to fetch. For example, the following statement defines the set of all Customer records:

```
FOR EACH Customer:
```

ABL allows you to define a set of records in a variety of ways. You define the set of records in a Record phrase. For more information on the syntax of Record phrases, see *ABL Reference*.

You can specify a Record phrase for the following ABL statements:

- FIND
- FOR [EACH]
- OPEN QUERY
- DO PRESELECT
- REPEAT PRESELECT

The examples that follow illustrate some of the flexibility that you have when defining a set of records. The Record phrases are highlighted. You can build complex Record phrases using the AND and OR operands.

This example defines a set of one record (Customer 11):

```
FIND Customer WHERE Customer.CustNum = 11.
```

This example uses the word-indexed field Comments to define a set of records (all Customer records containing the word "ship"):

```
FOR EACH Customer WHERE Customer.Comments CONTAINS "ship":
```

This example creates the subset of all **Order** records with the **Customer** number = 11:

```
OPEN QUERY ordqry FOR EACH Order WHERE Order.CustNum = 11.
```

This example defines a set of Customer records (those between 25 and 50) to be preselected:

```
REPEAT PRESELECT EACH Customer
  WHERE Customer.CustNum > 25 AND Customer.CustNum < 50:
```

Fetching records

After you define a set of records, ABL must fetch the records. How ABL fetches records depends in part on which statements you use to fetch the records. The following table summarizes the differences in record fetching between the FIND, FOR EACH, OPEN QUERY, and PRESELECT statements.

Table 4: Record fetching

Statement to define set of records	Statement to fetch the records	Structure used to locate records
FIND	FIND	Index
FOR EACH	FOR EACH	Results List or Index

Statement to define set of records	Statement to fetch the records	Structure used to locate records
OPEN QUERY	GET	Results List or Index
DO PRESELECT or REPEAT PRESELECT	FIND	Results List

Order of fetched records

The FOR EACH statement, OPEN QUERY statement, and PRESELECT option may use multiple indexes to satisfy a query. When multiple indexes are used, the order of returned records is not guaranteed. You can enforce an order by using the BY option.

The following example returns the selected Customer records in ascending CreditLimit order and within CreditLimit in Name order:

```
FOR EACH Customer BY Customer.CreditLimit BY Customer.Name:
```

Sample record fetches

The examples that follow show some of the many ways you can access records. These examples are not meant to be exhaustive, but merely to show some of the flexibility provided by ABL, as shown.

- Specifies the FIRST, LAST, NEXT, or PREV options to step through all records in a particular sequence:

```
FIND FIRST Customer.
```

- Specifies boolean expressions to describe the record or records to be fetched:

```
FOR EACH Customer
  WHERE Customer.CreditLimit > 5000 AND Customer.Balance < 12000:
```

- Specifies a constant value of the primary index for the record. This works only if the primary index is single-component. Also, this technique is not supported for the OPEN QUERY statement:

```
FIND Item 12.
```

- Specifies one or more field values that are currently in the screen buffer or record buffer:

```
PROMPT-FOR Customer.CustNum WITH FRAME abc.
FIND Customer USING FRAME abc Customer.CustNum.
DISPLAY Customer.Name.
```

- Specifies a previously found related record from another table (the two tables must share a field with the same name, and that field must have a UNIQUE index in at least one of the tables):

```
FIND FIRST Customer.
FOR EACH Order OF Customer: /* This uses the custnum field */
```

- Specifies a CONTAINS clause on a word-indexed field:

```
FOR EACH Customer WHERE Customer.Comments CONTAINS "ship":
```

You cannot use a CONTAINS clause with the FIND statement. You can use CONTAINS only with the OPEN QUERY and FOR EACH statements.

ROWID and RECID data types

ABL provides two structure types to support record fetches. One structure type, the index, you define in the schema of your database. The other structure type, a *results list*, is temporary; the AVM builds it at runtime. The results list associated with a DO, REPEAT, or OPEN QUERY statement with the PRESELECT option is sometimes called a *preselect list*.

In addition, there are two data types, ROWID and RECID, that allow you to retrieve a pointer to a fetched record. You can use this pointer to:

- Position to and retrieve a record from a results list
- Refetch a record and modify its lock status
- Store as a future record reference

In addition to the examples in this section, you can learn more about ROWID in *Develop ABL Applications*.

Comparing ROWID and RECID

ROWID is supported by all DataServers. Earlier Progress versions provide RECID as the only way to fetch a record pointer (supported in this version for backward compatibility). RECID is limited to a 4-byte record address supported by only a few DataServers and standard OpenEdge. ROWID provides a variable byte string that can represent a record address for any type of DataServer. For DataServers that use the 4-byte address supported by RECID (including OpenEdge), ROWID also uses a 4-byte value. Thus, there is no loss in performance using the more portable ROWID instead of RECID.

Returning record ROWID values

ABL provides a function named after the ROWID data type to return ROWID values. Given a buffer name, the ROWID function returns the ROWID of the current record in the buffer. This example fetches the first Customer record, and if it has a balance, refetches it to lock it for update:

```
DEFINE VARIABLE custrid AS ROWID NO-UNDO.

FIND FIRST Customer NO-LOCK.
custrid = ROWID(Customer).

IF balance > 0 THEN DO:
  FIND Customer WHERE ROWID(Customer) = custrid EXCLUSIVE-LOCK.
  UPDATE Customer.
END.
```

Storing and retrieving ROWID and RECID values

As shown in the previous example, you can store ROWID values in ROWID variables. You can also store them in work table fields. Thus, the following are valid ROWID storage definitions:

```
DEFINE VARIABLE wkrid AS ROWID NO-UNDO EXTENT 20.

DEFINE WORK-TABLE wtrid
  FIELD wkrid AS ROWID.
```

You cannot store ROWID values in database or temporary tables, but you can store their hexadecimal string representations using the STRING function. You can then retrieve the string as a ROWID value using the TO–ROWID function:

```
DEFINE TEMP-TABLE ttRid NO-UNDO
  FIELD ridfld AS CHARACTER.

FOR EACH Customer WHERE Customer.Balance = 0 NO-LOCK:
  CREATE ttRid.
  ASSIGN ttRid.ridfld = STRING(ROWID(Customer)).
END.

DO TRANSACTION:
  FOR EACH ttRid:
    FIND Customer WHERE ROWID(Customer) = TO-ROWID(ttRid.ridfld).
    DELETE Customer.
  END.
END.
```

You can store RECID values directly in a database or temporary table.

Additional ABL support

Several additional statements use ROWID and RECID values directly. For example, the REPOSITION statement sets the query position to a record given by its ROWID or RECID. For more information, see Results lists on page 54.

Also, because RECID is not supported by all DataServers, ABL provides the DBRESTRICTIONS function to indicate whether a particular DataServer supports it.

Converting from RECID to ROWID

When changing an application to use ROWID that currently uses RECID, you can complete the change with only a keyword substitution if your application does not:

- Reference RECID values as integers
- Store RECID values in database or temporary tables

Otherwise, after you change all RECID references to ROWID, you must rewrite your integer references to use character strings. If you use database or temporary tables, you must also convert the relevant fields to CHARACTER fields, and use the STRING and TO-ROWID functions to store and retrieve ROWID values. However, note that some DataServers build a string for a single ROWID that can reach up to several hundred bytes (including a complete WHERE clause).

All DataServer tables support ROWID references except those, such as views, that do not have unique row identifiers. DataServers from earlier Progress versions also support ROWID references. Progress Versions 7.3A and later use an internal RECID that transparently converts to a ROWID in the client.

Writing DataServer-portable applications

The least portable feature of ROWID references is the scope of a ROWID reference and when it changes for each DataServer. To maximize portability, follow these rules:

- Always assign values to unique keys before returning the ROWID value of a record. Some DataServers use a unique key to generate the ROWID value.
- If you UNDO a DELETE of a record for which you have stored the ROWID value, return the ROWID value again after the UNDO. It might be different after the record is recreated.
- If you update a unique key value for a record, return its ROWID again to replace any prior value.
- Never expect a record to have the same ROWID value between sessions.

Note that each DataServer uses a different method to generate a ROWID for a table, and sometimes for different tables in the same database. Therefore, never expect ROWID values to be identical, or even compatible, between otherwise duplicate tables from different DataServers.

For more information on ROWID value construction and scope for your DataServer, see your OpenEdge DataServer guide.

Results lists

A results list is a list of ROWIDs that satisfy a query. The results list allows you to quickly access the records in the record set you define, and allows ABL to make the records available one at a time, as needed.

When more than one row satisfies a query, the AVM doesn't lock all of the records and hold them until you release them. Instead, when a specific record is needed, the AVM uses the record's ID to go directly to the record, locking only that record. By going directly to the record, the AVM also ensures that you have the latest copy of the record.

When you open a query, the AVM does not normally build the entire results list. Instead it initializes the results list and adds to it as needed. The NUM-RESULTS function returns the number of records currently in the results list. This is not necessarily the total number of records that satisfy the query.

Whether and when the AVM builds the results list depends on the type of the query. As shown in Table 4: Record fetching on page 50, the DO or REPEAT PRESELECT statements always use a results list, while the FOR EACH and OPEN QUERY statements sometimes use a results list.

Queries have the following characteristics:

- **Scrolling versus non-scrolling** — A query is scrolling if you specify SCROLLING in the DEFINE QUERY statement or if you define a browse for the query. You can use the REPOSITION statement to change your current position within the results list. For a non-scrolling query, you can only move sequentially through the rows by using the FIRST, LAST, NEXT, and PREV options of the GET statement. Scrolling queries must use a results list (often initially empty); non-scrolling queries might not.
- **Index-sorted** — If the order of the rows in the query can be determined by using a single index, the query is *index-sorted*. For an index-sorted query, The AVM can use the index to order records without a results list. However, if the query requires additional sorting it must also be presorted.
- **Presorted** — If the query requires any sorting without an index or with multiple indexes, it must be *presorted*. For a presorted query, the AVM must read all the records, sort them, and build the complete results list before any records are fetched.
- **Preselected versus non-preselected** — You can force the AVM to build a preselected results list by specifying PRESELECT on the OPEN QUERY, DO, or REPEAT statement.

The following table summarizes how the results list is built for each type of query.

Table 5: Results lists for specific query types

Query type	Results list
Non-scrolling, index-sorted, no preselection	None.
Scrolling, no sorting, no preselection	Empty list[1] established when query is opened. Records are added to the results list as needed.
Presorted or preselected	Complete list built when query is opened.

There are two cases where the AVM has to build the entire results list when you first open the query:

- When you have explicitly used the PRESELECT option. In this case, the AVM performs a preselection phase in which it reads each record of the query. You can specify the lock type to use during the preselection phase. (For an OPEN QUERY, the lock type you specify in the OPEN QUERY statement is used for the preselection phase.) These locks are released immediately unless the preselection occurs within a transaction.
- When you have not used the PRESELECT option, but have specified sort criteria that cannot be performed using an index. In this case, the AVM performs a presort phase in which it reads each record of the query with NO-LOCK and builds the results list.

For example, the following statement explicitly uses the PRESELECT option and forces the AVM to build the entire results list immediately:

```
OPEN QUERY cust-query PRESELECT EACH Customer
  WHERE Customer.CreditLimit > 1500.
```

If you had used FOR instead of PRESELECT, the AVM would not have had to build the entire results list because it uses the primary index to fetch the records. It could use this index to find the first or last record for the query; it only needs to search forward or backward through the index until it finds a record that satisfies the WHERE clause.

You can use the PRESELECT option of the OPEN QUERY statement when you need to know immediately how many records satisfy the query or you can use it to immediately lock all the records that satisfy the query.

[1] If a browse is defined for the query, the results list initially contains one row.

The AVM also builds a complete results list when you open a query with a sort condition that cannot be resolved using a single index. Suppose you open a query on the Customer table as follows:

```
OPEN QUERY cust-query FOR EACH Customer BY Customer.City.
```

Because there is no index for the city field, the AVM must retrieve all the records that satisfy the query (in this case, all the Customer records), perform the sort, and build the entire results list before any records can be fetched. Until it performs this sort, the AVM cannot determine the first or last record for the query. If an index were defined on the city field, the AVM could use that index to fetch the records in sorted order (forwards or backwards) and would not need to build the results list in advance.

If the sort conditions for a query can be resolved using a single index, you can use the GET statement with the FIRST, LAST, NEXT, and PREV options on that query. For example, the following query is sorted using the primary index:

```
OPEN QUERY custqry FOR EACH Customer.
GET FIRST custqry.
DISPLAY Customer.CustNum Customer.Name.  /* Display first record */
PAUSE.
GET NEXT custqry.
DISPLAY Customer.CustNum Customer.Name.  /* Display second record */
PAUSE.
GET LAST custqry.
DISPLAY Customer.CustNum Customer.Name.  /* Display last record */
PAUSE.
GET PREV custqry.
DISPLAY Customer.CustNum Customer.Name.  /* Display second-to-last record */
```

Because the sorting is done with a single index, you can move freely forwards and backwards within the query.

Note: If you want to use the REPOSITION statement on a query, you must make the query scrolling by specifying the SCROLLING option in a DEFINE QUERY statement.

Navigating a results list

As shown in the Table 4: Record fetching on page 50 table, results lists are associated with the OPEN QUERY and GET statements. However, the AVM only guarantees a results list if you first define the query with the SCROLLING option. For example:

```
DEFINE QUERY custqry FOR Customer SCROLLING.
```

This option indicates to the AVM that you want to use the results list for multi-directional navigation.

You can use the REPOSITION statement to specify how many places forward or backward you want to move, so that you can skip over a given number records. It also allows you to move to a specific ROWID.

The REPOSITION statement changes your location in the results list but does not actually fetch the record (unless the query is associated with a browse widget). To actually fetch records in a results list, you use the GET statement. The following example illustrates how the REPOSITION statement works:

```
DEFINE VARIABLE rid AS ROWID NO-UNDO.   /* to save the ROWID of cust 4 */

DEFINE QUERY q FOR Customer SCROLLING.

OPEN QUERY q FOR EACH cust.
GET NEXT q.                 /* gets cust no. 1 */
GET NEXT q.                 /* gets cust no. 2 */
                            /* query is positioned ON cust 2 */
GET PREV q.                 /* gets cust no. 1 */
REPOSITION q FORWARD 0.     /* query is positioned BETWEEN cust 1 and 2 */
GET NEXT q.                 /* gets cust no. 2 */
                            /* query is positioned ON cust 2 */
REPOSITION q FORWARD 1.     /* query is positioned BETWEEN cust 3 and 4 */
GET NEXT q.                 /* gets cust no. 4 */
rid = ROWID(cust).          /* query is positioned ON cust 4 */
REPOSITION q BACKWARD 2.    /* query is positioned BETWEEN cust 2 and 3 */
GET PREV q.                 /* gets cust no. 2 */
REPOSITION q TO ROWID(rid). /* query is positioned BETWEEN cust 3 and 4 */ GET
NEXT q.                     /* gets cust no. 4 */
```

After a record is fetched (with a GET statement), the results list position is on the ROWID, so that GET NEXT gets the next record, and GET PREV gets the previous record. After a REPOSITION, the position is always **between** two records. Thus, REPOSITION FORWARD 0 repositions the results list immediately after the current record. GET NEXT fetches the next record; GET PREV fetches the previous record. REPOSITION FORWARD 1 repositions the results list between the next record and the record after it.

To find the total number of rows in a results list, you can use the NUM-RESULTS function. To find the current position within a results list, you can use the CURRENT-RESULT-ROW function.

As the Table 4: Record fetching on page 50 table shows, the AVM also creates results lists for FOR EACH statements and for DO and REPEAT statements with the PRESELECT phrase. However, you cannot navigate freely through a results list created for the FOR EACH statement. If the results list was created for the FOR EACH statement, then the AVM automatically steps through the results list in sorted order. For example:

```
FOR EACH Customer NO-LOCK BY Customer.Name:
  DISPLAY Customer.Name Customer.City Customer.State.
END.
```

Within a PRESELECT block, you can use the FIND statement to move backwards and forwards through the results list:

```
/* This code fragment displays all Customers in descending order */

DO PRESELECT EACH Customer NO-LOCK:
  FIND LAST Customer.    /* last position in list */
  DISPLAY Customer.CustNum Customer.Name WITH FRAME a DOWN.
  REPEAT:
    FIND PREV Customer.  /* move backward through list */
    DOWN WITH FRAME a.
    DISPLAY Customer.CustNum Customer.Name WITH FRAME a.
  END.
END.
```

FIND repositioning

After executing FOR EACH statements or FIND statements, the AVM might reposition subsequent FIND statements to the last record fetched (except for FIND statements occurring in PRESELECT blocks). For repositioning to occur, the same record buffer must be used. Also, repositioning after FOR EACH statements can differ between Progress Version 8.0B and Versions 8.0A and earlier, depending on the options you use.

Note: Repositioning does not occur for a subsequent FIND if the FIND specifies a unique key (that is, the FIND does not use the NEXT or PREV options).

Repositioning after FIND fetches

The AVM uses index cursors to keep track of what record you last fetched. This is important if you use the FIND statement to fetch a record. For example, depending on what was the last record fetched, the following statement returns a different record:

```
FIND NEXT Customer.
```

If you had last fetched the first Customer record, this statement would fetch the second Customer record. If you had just fetched the fourth Customer record, this statement would fetch the fifth Customer record.

A table can have multiple indexes, and the cursor position in each index dictates what the next record is in that index. For example, the following code fragment fetches Customers 1 and 21:

```
FIND FIRST Customer NO-LOCK.
DISPLAY Customer.CustNum Customer.Name Customer.Country Customer.PostalCode.
PAUSE.
FIND NEXT Customer NO-LOCK USE-INDEX country-post.
DISPLAY Customer.CustNum Customer.Name Customer.Country Customer.PostalCode.
```

In the country–post index, the **next** record after Customer 1 is Customer 21. the AVM uses the index cursor to establish the correct context.

Sometimes cursor repositioning is tricky. For example, the following code fragment returns Customer 6 and Customer 7 (you might expect Customer 6 and Customer 2):

```
FIND FIRST Customer NO-LOCK WHERE Customer.CustNum > 5.
DISPLAY Customer.CustNum Customer.Name.
PAUSE.
FIND NEXT Customer NO-LOCK WHERE Customer.CustNum > 1.
DISPLAY Customer.CustNum Customer.Name.
```

The first FIND statement causes the AVM to reposition the CustNum index cursor to point to Customer 6. The second FIND statement begins the search from that location, not from the beginning of the CustNum index.

Fetching field lists

When fetching records with a FOR EACH statement or query, the AVM typically retrieves all the fields of a record, whether or not your application needs them. This can have a costly impact on performance, especially when browsing records over a network.

ABL automatically optimizes preselected and presorted fetches from remote OpenEdge databases using field lists. A *field list* is a subset of the fields that define a record and includes those fields that the client actually requires from the database server. For preselected and presorted fetches, ABL can deduce this field list at compile time from the code.

You can also specify field lists explicitly for many types of AVM record fetches, including:

- Queries
- FOR statements
- DO PRESELECT statements
- REPEAT PRESELECT statements
- SQL SELECT statements

This section explains how to use field lists in ABL. For information on specifying field lists in SQL SELECT statements, see *Develop SQL for OpenEdge*.

Field list benefits

Field lists can provide significant performance benefits when:

- **Browsing over a network** — With the reduction in network traffic, tests show that field lists can increase the performance of fetches from both remote OpenEdge and DataServer databases by factors of from 2 to 10, depending on the record size and number of fields in the list.
- **Fetching from local DataServers** — Some local DataServers can yield significant improvements in fetch performance when only a portion of the record is read.
- **Fetching from remote DataServers that send multiple rows at a time** — DataServers that package multiple rows per network message show noticeable performance gains using field list fetches.

In general, these benefits mean that a multi-user query application can handle more network clients when fetching field lists than when fetching whole records. For more information on the availability and benefits of field list fetches with DataServers, see the OpenEdge DataServer guides.

Specifying field lists in ABL

In ABL, you can specify a field list in two contexts:

- Following each buffer name specified in a DEFINE QUERY statement
- Following the buffer name specified for fetching in each Record phrase of a FOR, DO PRESELECT, or REPEAT PRESELECT statement

This is the syntax for specifying a field list in all cases:

Syntax

```
record-bufname
  [ FIELDS [ ( [ field . . . ] ) ]
  | EXCEPT [ ( [ field . . . ] ) ]
]
```

The *record–bufname* reference specifies the table buffer you are using for the fetch, and a *field* reference specifies a field in the table. If *field* is an array reference, the whole array is fetched. The FIELDS form lists the fields included in the fetch, and the EXCEPT form lists the fields excluded from the fetch. FIELDS without *field* references fetches enough information to return the ROWID of a record, and EXCEPT without *field* references or *record–bufname* alone fetches a complete record.

Queries versus record selection blocks

For a query, you must specify the field lists in the DEFINE QUERY statement, not the Record Phrase of the OPEN QUERY statement. Thus, the following two procedures, i-fldls1.p and i-fldls2.p, are functionally equivalent.

i-fldls1.p

```
DEFINE QUERY custq FOR Customer FIELDS(Name CustNum Balance).
OPEN QUERY custq PRESELECT EACH Customer.
REPEAT:
  GET NEXT custq.
  IF AVAILABLE(Customer) THEN
    DISPLAY Customer.Name Customer.CustNum Customer.Balance.
  ELSE LEAVE.
END.
```

i-rldls2.p

```
REPEAT PRESELECT EACH Customer FIELDS (Name CustNum Balance):
  FIND NEXT Customer.
  DISPLAY Customer.Name Customer.CustNum Customer.Balance.
END.
```

Shared queries

If you specify field lists in a NEW SHARED query, the matching SHARED query definitions in external procedures only have to include the FIELDS or EXCEPT keyword as a minimum field list reference. The complete field list is optional in the SHARED query definitions, but required in the NEW SHARED query definition.

However, the AVM raises the ERROR condition when you run a procedure with a SHARED query if you:

- Specify a field list in the SHARED query to match a NEW SHARED query that has no field list.
- Do not specify a field list reference in the SHARED query to match a NEW SHARED query that has a field list.

Avoiding implied field list entries

Under certain conditions, the AVM adds fields to a specified field list when they are required by the client to complete the record selection. The most common case is when you specify a join condition with the OF option and you do not include the join field in the field list:

```
FOR EACH Customer FIELDS (Name),
  EACH Invoice FIELDS (InvoiceNum Amount) OF Customer:
  DISPLAY Customer.Name Customer.CustNum Invoice.InvoiceNum Invoice.Amount.
```

In this case, the AVM adds **Customer.CustNum** to the list because the client requires it to complete the join between the Customer and invoice tables.

However, never rely on implied entries in a field list to provide the fields you need. If you reference an unfetched field, the AVM raises the ERROR condition at runtime. Future versions of OpenEdge might change the criteria used to distribute record selection between the client and server. Thus in the previous example, if the server does not return Customer.CustNum to the client to complete the join, the DISPLAY statement executes with ERROR. This might happen, for example, if the DataServer you use or a future version of OpenEdge actually completes the specified join on the server.

Therefore, **always** specify all the fields you plan to reference in your field lists. There is no extra cost for specifying a field that the AVM can also add implicitly.

Updating and deleting with field lists

After fetching a field list, if your procedure updates the fetched record, the AVM always rereads the complete record before completing the update. In fact, if you fetch a field list with EXCLUSIVE-LOCK, the AVM reads the complete record anyway. This is to ensure proper operation of updates and the before-image (BI) file. (For information on BI files, see *Manage ABL Applications*.)

Also, if you delete a record after fetching a field list for it, the AVM rereads the complete record for the following cases:

- If you delete from an OpenEdge database, the AVM always re-reads the complete record.
- If you delete from a DataServer database, the AVM rereads the complete record if the delete occurs in a subtransaction, in order to create the local before-image (LBI) note. (For information on LBI files, see *Manage ABL Applications*.)

Thus, if you fetch with NO-LOCK or SHARE-LOCK, avoid using field lists if you expect to perform a high ratio of updates or deletes to fetches. For example, this is an inefficient construction with an OpenEdge database:

```
FOR EACH Customer FIELDS (Name Balance):
  DISPLAY "Deleting Customer" Customer.Name "with balance:" Customer.Balance.
  DELETE Customer.
END.
```

This procedure rereads the complete record for each field list that it fetches, and thus fetches twice per record. Without the field list, the same procedure fetches only once per record.

Updating and deleting with query field lists

For queries, especially those attached to a browse widget, there is little concern about updates and deletes, because the complete results list for a query is built before any updates or deletes take place. In this case, updates and deletes are selective over the entire query. Therefore, field lists can greatly enhance query performance, no matter how many updates or deletes a browse user completes.

Cursor repositioning and field lists

FOR and relative FIND statements reposition all open index cursors for the same buffer. (For more information, see FIND repositioning on page 58.) However, in order to reposition a buffer's index cursors, the AVM must have all the index fields available in the buffer. If you fetch a field list that excludes some of these fields, the AVM marks the relevant indexes as being incorrectly positioned.

This does not matter for the query or fetch loop that uses the field list, because the AVM might never reference the relevant indexes. However, if you later execute a FIND NEXT or FIND PREV using one of the badly positioned indexes, the AVM raises the ERROR condition and the FIND fails.

To avoid this error, always specify the fields in your field lists that participate in the indexes that you reference.

Field list handling in degenerate cases

When you specify field lists, the AVM is very flexible where it cannot make use of them. The AVM either returns complete records automatically or allows you to bypass field list processing completely for the following cases:

- **DataServers that do not support SHARE-LOCK** — If you execute a SHARE-LOCK fetch from a DataServer that does not support SHARE-LOCK, the AVM ignores the field list and retrieves the complete record. In some circumstances, the AVM must replace the fetched record with a new version. Because the SHARE-LOCK is meaningless, the new version can be different from the previous one, and the AVM requires the complete record to ensure that the user receives the correct data.

  Note: You can avoid SHARE-LOCK fetches with the help of the CURRENT-CHANGED function. For more information, see *Develop ABL Applications*.

- **Multiple queries returning the same record** — If you specify two queries that return two different field lists for the same record, the AVM cannot always consolidate the two lists. In that case, the AVM must reread the complete record. Such occurrences are rare, but they can impose a minor performance penalty with little impact on overall performance.

- **Deployment problems** — While programmers must ensure that their field lists are complete, run-time errors can still occur during application deployment. This is especially likely when a new database (schema) trigger is defined that references an unfetched field. To work around this type of problem, OpenEdge provides the Field List Disable (-fldisable) client session parameter. This is a run-time parameter that causes the AVM to ignore field lists in the r-code and fetch complete records. This might degrade performance, but allows the application to run until a fix can be made.

Thus, while using field lists can lead to difficulties, the AVM provides a way around most of them. And when used optimally, the performance gains can make field lists well worth the extra attention they might require.

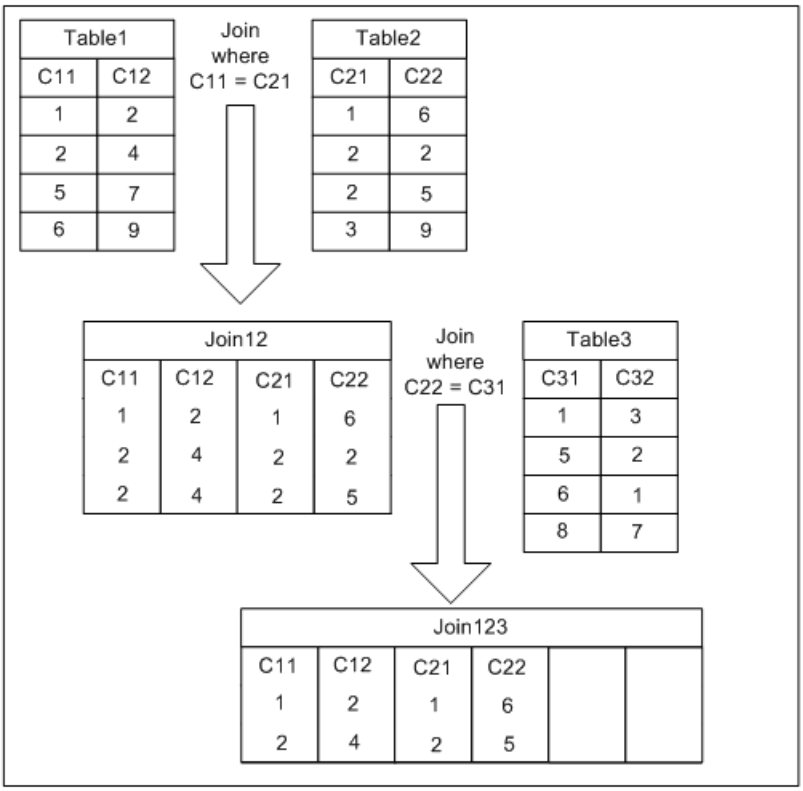
Joining tables

When you read from multiple tables using a single statement, such as a FOR EACH or OPEN QUERY statement, the AVM returns the results as a join of the tables. A *join* is a binary operation that selects and combines the records from multiple tables so that each result in the results list contains a single record from each table. That is, a single join operation combines the records of one table with those of another table or combines the records of one table with the results of a previous join.

```
FOR EACH Table1, EACH Table2 WHERE C11 = C21, EACH Table3 WHERE C22 = C31:
  DISPLAY C11 C12 C21 C22 C31 C32 WITH TITLE "Join123".
```

The following figure shows how you can join three tables.

Figure 3: Inner joins



A table or prior join can be either on the left- or righthand side of a join operation. Thus, the results of joining the three tables in the above figure depends on two join operations—one join between Table1 (left-hand side) and Table2 (righthand side) and one join between the first join (left-hand side) and Table3 (right-hand side). The relations C11 = C21 and C22 = C31 represent *join conditions*, conditions that determine how one table is related to the other (that is, which records selected from one table join with the records in the other table). How the records from joined tables are combined depends on the order of the tables in the join, the type of join operation used, and the selection criteria applied to each table.

Specifying joins in ABL

ABL supports two types of joins:

- **Inner join** — Supported in all statements capable of reading multiple tables, including the FOR, DO, REPEAT, and OPEN QUERY statements. An *inner join* returns the records selected for the table (or join) on the left side combined with the related records selected from the table on the right. For any records not selected from the right-hand table, the join returns no records from either the left or right sides of the join. Thus, only related records that are selected from both sides are returned for an inner join. The Figure 3: Inner joins on page 63 figure shows an example of inner joins.
- **Left outer join** — Supported only in the OPEN QUERY statement. A *left outer join* returns the records selected for an inner join. In addition, for each set of records selected from the table (or join) on the left side, a left outer join returns the Unknown value (?) from the table on the right where there is no record selected or otherwise related to the records on the left. That is, records from the left-hand table (or join) are preserved for all unmatched records in the right-hand table. The Figure 4: Left outer joins on page 65 figure shows an example of left outer joins using the same tables as in the Figure 3: Inner joins on page 63 figure.

Note: For Progress versions earlier than Version 8, only the inner join is supported.

Specifying the type of join

The Record phrase that specifies the right-hand table of a join also indicates the type of join operation. A Record phrase specifies an inner join by default. To specify a left outer join, you include the [LEFT]OUTER-JOIN option anywhere in the Record phrase. Where you specify a list of multiple Record phrases in a record-reading statement, the join logic allows you to specify only one set of contiguous inner joins at the beginning (left side) of the list and one set of contiguous left outer joins at the end (right side) of the list. Each right-hand Record phrase of a left outer join must contain the OUTER-JOIN option up to and including the last left outer join in the list. For more information, see Mixing inner and left outer joins on page 68.

Relating and selecting tables

You can specify join conditions (table relations) using the OF option or WHERE option of the Record phrase that specifies the join. The OF option specifies an implicit join condition based on one or more common field names in the specified tables. The common field names must participate in a unique index for at least one of the tables. The WHERE option can specify an explicit join based on any field relations you choose, and you can use this option further to specify selection criteria for each table in the join. For an inner join, if you do not use either option, the AVM returns a join of all records in the specified tables. For a left outer join, you must relate tables and select records using the OF option, the WHERE option, or both options.

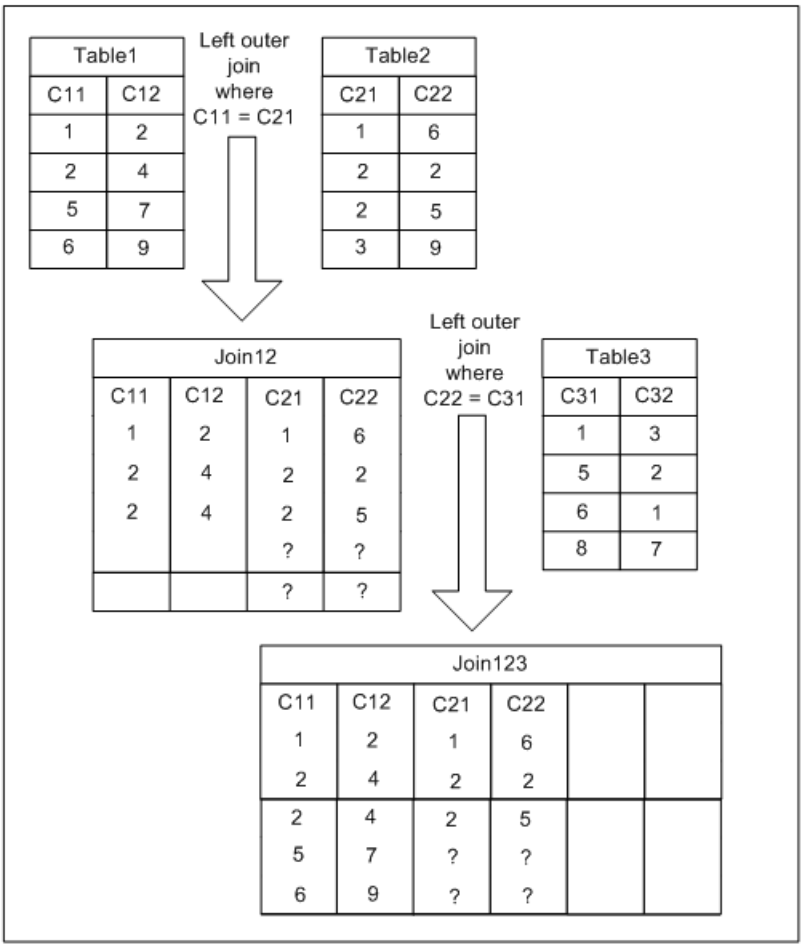
Note: Work tables and temporary tables can also participate in joins. However, work tables do not have indexes. So, if you specify join conditions using the OF option with a work table, the other table in the join must be a database or temporary table with a unique index for the fields in common. For more information on work tables and temporary tables, see *Develop ABL Applications*.

The following code fragment generates the left outer joins shown in the following figure:

```
DEFINE QUERY q1 FOR Table1, Table2, Table3.

OPEN QUERY q1 FOR EACH Table1, EACH Table2 OUTER-JOIN WHERE C11 = C21,
  EACH Table3 OUTER-JOIN WHERE C22 = C31.
GET FIRST q1.
DO WHILE AVAILABLE(Table1):
  DISPLAY C11 C12 C21 C22 C31 C32 WITH TITLE "Join123".
  GET NEXT q1.
END.
```

Note that the Record phrase for the right-hand table of each join specifies the OUTER-JOIN option. As the following figure shows, the primary benefit of a left outer join is that it returns every record on the left-hand side, whether or not related data exists on the right.

Figure 4: Left outer joins



Using joins instead of nested reads

Using joins provides an opportunity for the AVM to optimize the retrieval of records from multiple related tables using the selection criteria you specify. When you perform a nested read, for example using nested FOR EACH statements for different tables, you are actually implementing a join in an ABL procedure. However, by specifying one or more contiguous joins in a single FOR EACH statement or in the PRESELECT phrase of single DO or REPEAT statement, you minimize the complexity of your ABL code and leave the complexity of joining tables to the ABL interpreter.

For a single ABL query (OPEN QUERY statement), there is no other way to retrieve data from multiple tables except by using joins. With both inner and left outer join capability, you can use the OPEN QUERY statement to implement most queries that are possible using nested FOR EACH, DO, or REPEAT statements. As such, query joins provide the greatest opportunity for optimized multi-table record retrieval in ABL. Also, because browse widgets read their data from queries, you must use query joins to display multiple related tables in a browse.

However, use nested FOR EACH, DO, and REPEAT blocks wherever you require much finer control over how you access and manipulate records from multiple tables.

Using inner joins

The inner join is the default join type in a multi-table read or query. Use this type of join where you are only concerned with the data on the left side of the join for which there is related data on the right. For example, you might want to see only those Customers whose purchase of a single item comes close to their credit limit.

The query in `i-join1.p` performs three inner joins on the `sports2000` database to return this data.

i-join1.p

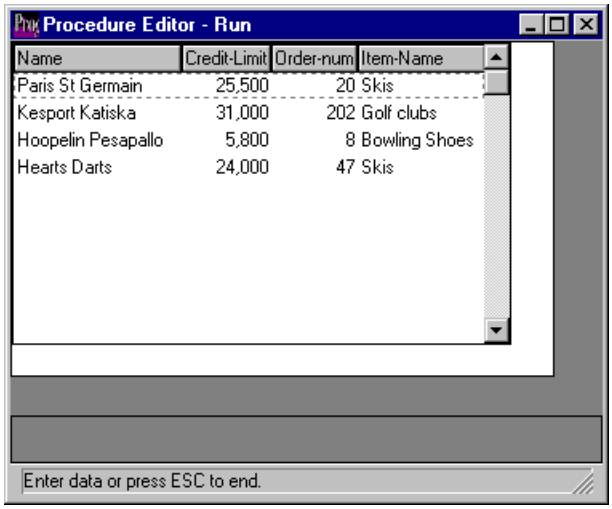
```
DEFINE QUERY q1 FOR Customer, Order, OrderLine, Item.
DEFINE BROWSE b1 QUERY q1
  DISPLAY Customer.Name Customer.CreditLimit Order.OrderNum Item.ItemName
  WITH 10 DOWN.

OPEN QUERY q1 PRESELECT EACH Customer,
/*1*/ EACH Order OF Customer,
/*2*/ EACH OrderLine OF Order
        WHERE (OrderLine.Price * OrderLine.Qty) >
              (.667 * Customer.CreditLimit),
/*3*/ EACH Item OF OrderLine.

ENABLE b1 WITH SIZE 68 BY 10.
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

These three items correspond to the `/*1*/`, `/*2*/`, and `/*3*/` that label the joins in the `i-join1.p` example:

1. To each **Customer**, join all related **Order** records.
2. To each **Order** in the previous join, join each related **OrderLine** record whose total purchase is greater than two-thirds the Customer's credit limit.
3. To each **OrderLine** in the previous join, join the related **Item** record to get the item name.

When executed, you get the output shown in the following figure. Thus, the relation of **Order** to **Customer** and the selection criteria on **OrderLine** reduces the total number of query rows from thousands of possible rows to four.

Figure 5: Inner join example



Using left outer joins

A left outer join is useful where you want to see all the data on the left side, whether or not there is related data on the right. For example, you might want to see the proportion of Customers who are ordering close to their credit limit as against those who are not.

The query in `i-join2.p` is identical to the one in `i-join1.p` (see Using inner joins on page 66) except that all the joins are left outer joins instead of inner joins.

i-join2.p

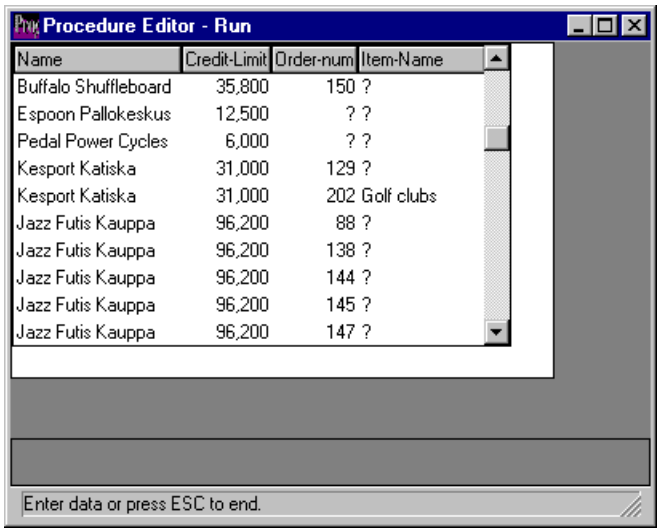
```
DEFINE QUERY q1 FOR Customer, Order, OrderLine, Item.
DEFINE BROWSE b1 QUERY q1
  DISPLAY Customer.Name Customer.CreditLimit Order.OrderNum Item.ItemName
  WITH 10 DOWN.

OPEN QUERY q1 PRESELECT EACH Customer,
/*1*/ EACH Order OUTER-JOIN OF Customer,
/*2*/ EACH OrderLine OUTER-JOIN OF Order
        WHERE (OrderLine.Price * OrderLine.Qty) >
              (.667 * Customer.CreditLimit),
/*3*/ EACH Item OUTER-JOIN OF OrderLine.

ENABLE b1 WITH SIZE 68 BY 10.
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

Thus, you see all Customers, whether or not they order close to their credit limit. These three items correspond to the `/*1*/`,`/*2*/`, and `/*3*/` that label the joins in the `i-join2.p` example:

1. To each **Customer**, join all related **Order** records, or join a null **Order** record if the Customer has no orders.
2. To each **Order** in the previous join, join each related **OrderLine** record whose total purchase is greater than two–thirds the Customer's credit limit, or join a null **OrderLine** record if all item purchases are less than or equal to two–thirds the Customer's credit limit. Also, join a null **OrderLine** record if the order is null (the Customer has no orders).
3. To each **OrderLine** in the previous join, join the related **Item** record to get the item name, or join a null **Item** record if the **OrderLine** is null (no **Order** or selected purchase for that Customer).

When executed, you get the output shown in the following figure. In this example, you see the same golf club order as in the Figure 5: Inner join example on page 66 figure along with many other orders that do not meet the selection criteria and some Customers who have no orders at all.

Figure 6: Left outer join example



Implementing other outer joins

In addition to left outer joins, there are right and full outer joins. A *right outer join* reverses the join order for the same tables joined with a left outer join. In ABL, you can implement a right outer join by doing a left outer join with the tables in reverse order, but leaving the order of displayed fields the same as for the left outer join. Thus, the `Unknown value (?)` from the right side appear on the left side of each displayed row, as if the tables were joined from right to left.

A *full outer join* combines the results of a left and right outer join into a single join. This is rarely used, but you can implement a full outer join by building one temporary table from the results of both a left and right outer join. For more information on temporary tables, see *Develop ABL Applications*.

Mixing inner and left outer joins

You might want to mix inner and left outer joins in order to filter and reduce the amount of data you need on the left side of your left outer joins. When mixing these two types of join, keep in mind that the last inner join in a query forces the results of all prior joins in the query to be inner joins. This is because any rows that contain the Unknown value (?) from a prior left outer join are eliminated by the following inner join. The effect is that the work of the prior left outer joins is wasted, and the same result is achieved as with contiguous inner joins, but much less efficiently. Therefore, in any query, keep your inner joins contiguous on the left with any left outer joins contiguous on the right.

Server-side join processing

Server-side join is a feature in which certain queries involving joins, execute on the database server, rather than the client. Server-side join improves performance by reducing the amount of data transferred across the network and by reducing the computational cost on the client.

Background

Prior to OpenEdge 12.0, when the AVM executed a join of multiple tables, some records which did not satisfy all levels of the join may have been returned to the AVM client, which needed to perform additional processing to eliminate these extra records from consideration. This methodology provided a correct response to the ABL application but with performance and security challenges. Starting in 12.0, incremental functionality was added to perform joins of multiple tables on the database server. This functionality is known as server-side join.

Requirements for server-side join use

The following requirements must be met for a join query to be executed server-side:

- The database server must be multi-threaded.
- All tables to be joined must be in the same logical database.
- There must be ten or fewer tables to be joined.
- `NO-LOCK` must be used.
- The ABL query construct must be one of the following:
  - `FOR` statement.
  - Static or dynamic query that does not use the `INDEXED-REPOSITION` option or involve an outer join.

Note: Although the query needs to be `NO-LOCK` for server-side join to be in effect, you can obtain records with a lock using the `GET` statement for a static query, or the desired `GET` method (for example, `GET-NEXT( )`) for a dynamic query.

Restrictions

If all the requirements are met, there may still be some cases where server-side join is not used.

- Server-side join is not used for queries with the following constructs on any child table:
  - User-defined functions, methods, or properties with a `GET` implementation.
  - `CAN-DO` function.
  - Expressions with MEMPTR and LONGCHAR data types.
  - `TENANT-WHERE` clause.
- The use of `RECID` and `ROWID` to join records disables server-side join.
- If the query contains a `BY` clause and the index used for the query provides the same sorting order specified by the `BY` clause, the join is done on the server. However, if the table does not contain any indexes that provide the sorting specified by the `BY` clause, and the `BY` clause does not specify only fields for the most inner table of the join, then the join is done on the client.

  Note: You can tell whether the sort was done by index or not by looking at the "Client Sort:" entry in the query information log file (see Query plan). "Client Sort: Y" indicates a non-index sort and "Client Sort: N" indicates the sort was done by index.

The following table shows different query examples and whether the join is executed on the server (On) or the client (Off).

Query example	Server-side join	Sort by index	Notes

`FOR EACH Customer NO-LOCK,`
`  EACH Order OF Customer NO-LOCK,`
`  EACH OrderLine NO-LOCK OF Order`
`  BY OrderLine.Price :`
 	
`DISPLAY`
`  Customer.NAME OrderLine.Price Order.Carrier.`
`END.`
	On	No	Using the BY clause on the last table's field – server-side join is always turned on.

`FOR EACH Customer NO-LOCK,`
`  EACH Order OF Customer NO-LOCK,`
`  EACH OrderLine NO-LOCK OF Order`
`  BY Customer.Address :`
 	
`DISPLAY`
`  Customer.NAME OrderLine.Price Order.Carrier.`
`END.`
	Off	No	Using the BY clause on the top parent table's field that results in non-index sorting causes server-side join to be turned off.

`FOR EACH Customer NO-LOCK,`
`  EACH Order OF Customer NO-LOCK,`
`  EACH OrderLine NO-LOCK OF Order`
`  BY Customer.NAME :`
 	
`DISPLAY`
`  Customer.NAME OrderLine.Price Order.Carrier.`
`END.`
	On	Yes	Using the BY clause on the top parent table field that results in index sorting causes server-side join to be turned on.

`FOR EACH Customer NO-LOCK,`
`  EACH Order OF Customer NO-LOCK,`
`  EACH OrderLine NO-LOCK OF Order`
`  BY  Order.Carrier :`
 	
`DISPLAY`
`  Customer.NAME OrderLine.Price Order.Carrier.`
`END.`
	Off	No	Using the BY clause on any middle table field that results in non-index sorting causes server-side join to be turned off.

Server-side join (-ssj) startup parameter

You may want to run some queries with server-side join enabled and disabled, to compare differences in performance. Server-side join can be disabled by using the Server-side join (-ssj) database server startup parameter, when you start the database up. You specify a value of "0" to disable it:

```
-ssj 0
```

Logging

To see if a particular query was done in the server, you can enable Query Information Logging for your client session. For example, start the client session using these startup parameters:

```
-clientlog logfilename
-logentrytypes QryInfo
-logginglevel 3
```

If the query was resolved in the database server, you see "Server-side join" in the log file entry.

For example, for a FOR statement, you see:

```
Type: FOR Statement, Server-side join
```

For a static query, you see:

```
Type: Statically Opened Query, Server-side join
```

For a dynamic query, you see:

```
Type: Dynamically Opened Query, Server-side join
```

The CONTAINS operator

You can use the CONTAINS operator of the WHERE option of the record phrase to query a OpenEdge database. For example, the following query displays each Item record of the sports2000 database whose CatDescription field contains the string "hockey":

```
FOR EACH Item WHERE Item.CatDescription CONTAINS "hockey":
  DISPLAY Item.
END.
```

Using the CONTAINS operator involves word indexes and word-break tables. This section explains why and tells you how to use word indexes and word-break tables. Specifically, the section covers:

- Using word indexes
- Using word-break tables
- Using the CONTAINS operator
- Word indexing external documents
- Word indexing non-OpenEdge databases

Note: Word indexes and word-break tables have international implications. To understand these implications, first read this section, then see *Internationalize ABL Applications*.

The CONTAINS operator and word indexes

For the AVM to process a query that uses the CONTAINS operator, the field mentioned in the WHERE option must participate in a word index. For example:

```
FOR EACH Item WHERE Item.CatDescription CONTAINS "hockey":
  DISPLAY Item.
END.
```

The AVM looks for the word indexes associated with the Item record and its CatDescription field. If no such word indexes are found, the AVM reports a run-time error. Otherwise, the AVM uses the word indexes to retrieve the Item records whose CatDescription field contains the string hockey.

Word indexes and word-break tables

In order for the AVM to use word indexes, it must first build and maintain them, which it does as you add, delete, and modify records that have fields that participate in them. Consider the sports2000 database's Item table, whose CatDescription field participates in a word index. Every time an Item record is added, the AVM examines the contents of the CatDescription field, breaks it down into individual words, and, for each individual word, creates or modifies a word index. To break down the contents of a field into individual words, the AVM must know which characters act as word delimiters. To get this information, the AVM consults the database's word-break table, which lists characters and describes the word-delimiting properties of each.

Creating and viewing word indexes

To define a new word index or to view the word indexes already defined on a database table, use the OpenEdge Data Dictionary utility. For example, you can use this utility to view the word indexes of the sports2000 database, including the word index associated with the Item table and its CatDescription field. For more information on creating and viewing word indexes using the OpenEdge Data Dictionary utility, see the OpenEdge Data Dictionary utility online help.

Word indexing very large fields

If a word index is defined on a CHARACTER field that is extremely large, you might have to increase the value of the Stash Area (–stsh) startup parameter. Doing so increases the amount of space OpenEdge uses to temporarily storage of modified indexes. For more information, see the reference entry for the Stash Area (–stsh) startup parameter in *Startup Command and Parameter Reference*.

Word indexing double-byte and triple-byte code pages

You can use word indexing with double-byte and triple-byte code pages. For more information, see *Internationalize ABL Applications*.

Using word-break tables

You can create word-break tables that specify word separators using a rich set of criteria. To specify and work with word-break tables involves:

- Specifying word delimiter attributes
- Understanding the syntax of word-break tables

- Compiling word-break tables
- Associating compiled word-break tables with databases
- Rebuilding word indexes
- Providing access to compiled word-break tables

Specifying word delimiter attributes

As mentioned previously, to break down the contents of a word-indexed field into individual words, the AVM needs to know which characters delimit words and which do not. The distinction can be subtle and sometimes depends on context. For example, consider the function of the dot in the character strings in the following table.

Table 6: Is the dot a word delimiter?

Character string	Function of the dot	Is the dot a word delimiter?
"Balance is $25,125.95"	Decimal separator	No
"Shipment not received.Call customs broker"	Period at end of sentence	Yes

In the first character string, the dot functions as a decimal point and does not divide one word from another. Thus, you can query on the word "$25,125.95." In the second character string, by contrast, the dot functions as a period, dividing the word "received" from the word "call."

To help define word delimiters systematically while allowing for contextual variation, ABL provides eight word delimiter attributes, which you can use in word-break tables. The eight word delimiter attributes appear in the following table.

Table 7: Word delimiter attributes

Word delimiter attribute	Description	Default
`LETTER`	Always part of a word	Assigned to all characters that the current attribute table defines as letters.In English, these are the uppercase characters A–Z and the lowercase characters a–z.
`DIGIT`	Always part of a word	Assigned to the characters 0–9.
`USE_IT`	Always part of a word	Assigned to the following characters: • Dollar sign ($) • Percent sign (%) • Number sign (#) • At symbol (@) • Underline (_)

Word delimiter attribute	Description	Default
BEFORE_LETTER	Part of a word only if followed by a character with the LETTER attribute; otherwise, treated as a word delimiter	–
BEFORE_DIGIT	Treated as part of a word only if followed by a character with the DIGIT attribute	Assigned to the following characters: • Period (.) • Comma (,) • Hyphen (–) For example, "12.34" is one word, but "ab.cd" is two words.
BEFORE_LET_DIG	Treated as part of a word only if followed by a character with the LETTER or DIGIT attribute	–
IGNORE	Ignored	Assigned to the apostrophe ('). For example, "John's" is equivalent to "Johns."
TERMINATOR	Word delimiter	Assigned to all other characters.

Understanding the syntax of word-break tables

Word delimiter attributes form the heart of word break tables, and you specify them using the following syntax:

Syntax

```
[ #define symbolic-name symbol-value ]...

[ Version = 9
  Codepage = codepage-name
  wordrules-name = wordrules-name
  type = table-type
]

word_attr =
{
  { char-literal | hex-value | decimal-value } , word-delimiter-attribute
     [ , { char-literal | hex-value | decimal-value }
          , word-delimiter-attribute ]... } ;
```

symbolic-name

The name of a symbol. For example: DOLLAR-SIGN

symbol-value

The value of the symbol. For example: "$'

Note: Although some versions of ABL let you compile word-break tables that omit all items within the second pair of square brackets, Progress Software Corporation (PSC) recommends that you always include these items. If the source-code version of a compiled word-break table lacks these items, and the associated database is not so large as to make this unfeasible, PSC recommends that you add these items to the table, recompile the table, reassociate the table with the database, and rebuild the indexes.

---

codepage-name

The name, not surrounded by quotes, of the code page the word-break table is associated with. The maximum length is 20 characters. For example: UTF–8

wordrules-name

The name, not surrounded by quotes, of the compiled word-break table. The maximum length is 20 characters. For example: utf8sample

table-type

The number 2.

Note: Some earlier versions of Progress allow a table type of 1. Although this is still supported, Progress Software Corporation (PSC) recommends, if feasible, that you change the table type to 2, recompile the word-break table, reassociate it with the database, and rebuild the indexes.

---

char-literal

A character within single quotes or a *symbolic–name,* which represents a character in the code page. For example: '#'

hex-literal

A hexadecimal value or a *symbolic–name,* which represents a character in the code page. For example: 0xAC

decimal-literal

A decimal value or a *symbolic–name*, which represents a character in the code page. For example: 39

word-delimiter-attribute

In what context the character is a word delimiter. You can use one of the following:

- LETTER
- DIGIT
- USE_IT
- BEFORE_LETTER
- BEFORE_DIGIT

- BEFORE_LET_DIG
- IGNORE
- TERMINATOR

Examples of word-break tables

The following is an example of a word-break table for Unicode:

```
/* a word-break table for Unicode */

#define DOLLAR-SIGN '$'

Version = 9
Codepage = utf-8
wordrules-name = utf8sample
type = 2

word_attr =
{  '.',         BEFORE_DIGIT,
   ',',         BEFORE-DIGIT,
   0x2D,        BEFORE_DIGIT,
   39,          IGNORE,
   DOLLAR-SIGN, USE-IT,
   '%',         USE-IT,
   '#',         USE_IT,
   '@',         USE-IT,
   '_',         USE_IT,
};
```

As the preceding example illustrates, word-break tables can contain comments delimited as follows:

```
/* this is a comment */
```

For more examples, see the word-break tables that ABL provides in source-code form. They reside in the `DLC/prolang/convmap` directory and have the file extension `.wbt`.

Note: ABL supplies a word-break table for each code page it supports.

Compiling word-break tables

After you create or modify a word-break table, you must compile it with the `PROUTIL` utility. The syntax is as follows:

Operating system	Syntax
UNIX Windows	`proutil -C wbreak-compiler` *`src-file rule-num`*

`src-file`

The name of the word-break table file to be compiled.

rule-num

A number between 1 and 255 inclusive that identifies this word-break table within your OpenEdge installation.

The PROUTIL utility names the compiled version of the word-break table proword.*rule–num*. For example, if *rule–num* is 34, PROUTIL names the compiled version proword.34.

Associating compiled word-break tables with databases

After you compile a word-break table, you must associate the compiled version with a database using the PROUTIL utility. The syntax is as follows:

Operating system	Syntax
UNIX Windows	proutil *database* -C word-rules *rule-num*

database

The name of the database.

rule-num

The value of *rule-num* you specified when you compiled the word-break table. To associate the database with the default word-break rules, set *rule-num* to zero.

Note: Setting *rule-num* to zero associates the database with the default word-break rules for the current code page. For more information on code pages, see *Internationalize ABL Applications*.

Word-break tables for double-byte and UTF-8 databases

OpenEdge ships compiled word-break tables for double-byte and UTF-8 databases. The word-break tables are pre-applied to the appropriate empty databases. This means that word-indexing is available without any additional work for those databases.

The compiled word-break tables are proword.245 through proword.254. The tables are in your *install-dir*/prolang/*language* directory or subdirectory with the corresponding empty database.

If you already have a proword file that uses the same number as one of these new tables, you should change its number to a number less than 240. You can then apply the new word rule to your existing database. This change does not require an index build.

Rebuilding word indexes

For word indexing to work as expected, the word-break table ABL uses to write the word indexes (to add, modify, or delete a record that contains a word index) and the word-break table the AVM uses to read word indexes (to process a query that contains the CONTAINS operator) must be identical. To ensure this, when you associate the compiled version of a word-break table with a database, the AVM writes cyclical redundancy check (CRC) values from the compiled word-break table into the database. When you connect to the database, the AVM compares the CRC values in the database to the CRC value in the compiled version of the word-break table. If they do not match, the AVM displays an error message and terminates the connection attempt.

If a connection attempt fails and you want to avoid rebuilding the indexes, you can try associating the database with the default word-break rules.

Note: This might invalidate the word indexes and require you to rebuild them anyway.

To rebuild the indexes, you can use the PROUTIL utility with the IDXBUILD or IDXFIX qualifier.

The basic syntax of PROUTIL with the IDXBUILD qualifier is:

Operating system	Syntax
UNIX Windows	`proutil db-name -C idxbuild [ all ] [ -T dir-name ] [ -TB blocksize ] [ -TM n ] [ -B n ]`

The basic syntax of PROUTIL with the IDXFIX qualifier is:

Operating system	Syntax
UNIX Windows	`proutil db-name -C idxfix`

Both the IDXBUILD or IDXFIX qualifiers have extensive options for selecting indexes and performance tuning. For the complete syntax of the PROUTIL utility, see *Manage the OpenEdge Database*.

Providing access to the compiled word-break table

To allow database servers and shared-memory clients to access the compiled version of the word-break table, it must reside either in the OpenEdge installation directory or in the location pointed to by the environment variable PROWD*rule–num*. For example, if the compiled word-break table has the name proword.34 and resides in the DLC/mydir/mysubdir directory, set the environment variable PROWD34 to DLC/mydir/mysubdir/proword.34.

Note: Although the name of the compiled version of the word-break table has a dot, the name of the corresponding environment variable does not.

Writing queries using the CONTAINS operator

Once you associate a compiled word-break table with a database that has word indexes, if necessary, populate the database and rebuild the word indexes. You can then write queries that use the CONTAINS operator.

Syntax of the CONTAINS operator of the WHERE option

The CONTAINS operator has the following syntax in the WHERE option of the record phrase:

Syntax

```
WHERE field CONTAINS string-expression
```

field

A field or array of type CHARACTER that participates in a word index.

string-expression

An expression of type CHARACTER that represents possible contents of *field*. The syntax of *string–expression* is as follows:

```
"word[ [ & | | | ! | ^ ] word ]..."
```

word

The word to search for.

The ampersand (&) represents logical AND, while the vertical bar (|), the exclamation point (!), and the caret (^) represent logical OR. AND limits your search to records that contain all words you specify, while OR enlarges your search to include any word you specify. You can combine ANDs and ORs within *string–expression*. You can also group items with parentheses to create complex search conditions.

You can use a wild card on the end of a string. For example, the string "sales*" represents "sales," "saleswoman," "salesman," "salesperson," and similar strings.

You can also define a character variable and assign a value to that variable.

Examples of the CONTAINS operator

Now that you know the syntax of the CONTAINS operator, you can write queries that use it.

The following query, which displays all Item records whose CatDescription field contains the word "hockey," demonstrates the CONTAINS operator in its simplest form:

```
FOR EACH Item WHERE Item.CatDescription CONTAINS "hockey":
  DISPLAY Item.
END.
```

The following is the equivalent query in SQL, which also allows CONTAINS:

```
SELECT * FROM Item
  WHERE CatDescription CONTAINS "hockey".
```

A CONTAINS string can contain multiple words connected by the AND operator (AND or &) and the OR operator (OR, |, or ^), optionally grouped by parentheses. For example:

```
...CONTAINS "free | gratis | (no & charge)"...
```

Note: The AND operator takes precedence over the OR operator. To override this default, use parentheses. Using parentheses can also make the text of a query clearer.

A CONTAINS string containing multiple contiguous words, such as:

```
...CONTAINS "credit hold"...
```

Is equivalent to a CONTAINS string containing multiple words connected by AND, such as:

```
...CONTAINS "credit AND hold"...
```

If a CONTAINS string contains multiple words, the order of the words is not significant. To retrieve records in a specific order, use the CONTAINS operator with the MATCHES operator. The following WHERE clause retrieves records whose comments field contains the words "credit" and "hold" in that order, perhaps with other words in between:

```
...WHERE Customer.Comments CONTAINS "credit hold"
     AND Customer.Comments MATCHES "*credit*hold*"...
```

Word indexes are case insensitive unless a field participating in the word index is case sensitive. The following two WHERE clauses are equivalent:

```
...WHERE Customer.Comments CONTAINS "CREDIT HOLD"...
```

```
...WHERE Customer.Comments CONTAINS "credit hold"...
```

You can combine CONTAINS with other search criteria, as in the following WHERE clause, which searches for records whose city field is Boston and whose comments field contains the word "credit" and either the word "hold" or "watch":

```
...WHERE Custeomer.City = "Boston"
    AND Customer.Comments CONTAINS "credit (hold ^ watch)"...
```

The following example demonstrates the use of a variable with the CONTAINS operator within the WHERE clause:

```
DEFINE VARIABLE search_wrd AS CHARACTER NO-UNDO INITIAL "The".

FOR EACH Customer NO-LOCK WHERE Customer.Name CONTAINS search_wrd:
  DISPLAY Customer.CustNum Customer.Name.
END.
```

Word indexing external documents

To create a word index on an existing document, import the text into a OpenEdge database, then index the text by line or by paragraph.

Indexing by line

To index a set of documents by line, you might create a table called line with three fields: document_name, line_number, and line_text. Define the primary index on document_name and a word index on line_text. Next, write a text-loading OpenEdge program that reads each document and creates a line record for each line in the document. To decrease the amount of storage required by the line table and to normalize the database, you might replace its document_name field with a document_number field, and create a document table to associate a document_name with each document_number.

When base documents change, you must update the line index. You can store a document ID as part of the record for each line. When a document changes, you can delete all lines with that document ID and reload the document.

The following program queries the line table using the word index:

```
DEFINE VARIABLE words AS CHARACTER NO-UNDO FORMAT "x(60)"
  LABEL "To find document lines, enter search words".

REPEAT:
  UPDATE words.
  FOR EACH line WHERE line_text CONTAINS words:
    DISPLAY line.
  END.
END.
```

The example prompts for a string of words, then displays each line that matches the search criteria.

Indexing by paragraph

Instead of indexing by line, you can index by paragraph. The technique resembles line indexing, but the text from a paragraph can be much longer. You can use paragraph indexes the same way you use line indexes. You can also index by chapter, by page, and by other units of text. The only difference is how your text-loading program parses the document into character fields. Otherwise, your word search code, as in the line table example, can be identical.

Word indexing non-OpenEdge databases

OpenEdge DataServers do not support word indexing. To create a word index on text from a non-OpenEdge database accessed by a DataServer that does not support word indexing, you must write an ABL routine to read records from the non-OpenEdge database and copy the text into a table in an OpenEdge database.

You might define an OpenEdge table with the fields `nonprog_primary_key` and `nonprog_text`. Define a word index on `nonprog_text`, then load text and keys from the non-OpenEdge database into the table. Then, use the word index to find the primary keys for records that contain specific words.

Sequences

Sequences are database objects that provide incremental values to an application. They can generate sequential values within any range of an ABL integer (–2,147,483,648 to 2,147,483,647) and with your choice of increment (positive or negative).

You can also define sequences to cycle or terminate when a specified limit is reached. A *cycling sequence* restarts after it reaches the specified limit, providing non-unique values for the life of the sequence. A *terminating sequence* stops incrementing after it reaches the specified limit, providing unique values for the life of the sequence as long as you do not explicitly reset it.

Using a sequence, for example, you can generate a unique series of key values automatically. ABL allows you to guarantee that the key value is unique (for terminating sequences) and created in a specified order. You can also use a sequence to generate an audit trail that creates sequential audit records in the same order they are written to the database.

You can also perform the same kinds of operations by using a *control table* instead of sequences. A control table is a database table that contains a single record with one or more fields. Each field holds a piece of control information such as the name of the application, the company that owns the application, and the last-used value for each unique integer ID in the database. For example, the control table might have fields that hold the last Customer number used and the last order number used. Each time a new Customer record is created, the record is read from the control table and the last Customer number is incremented. This number is then used in the new Customer record.

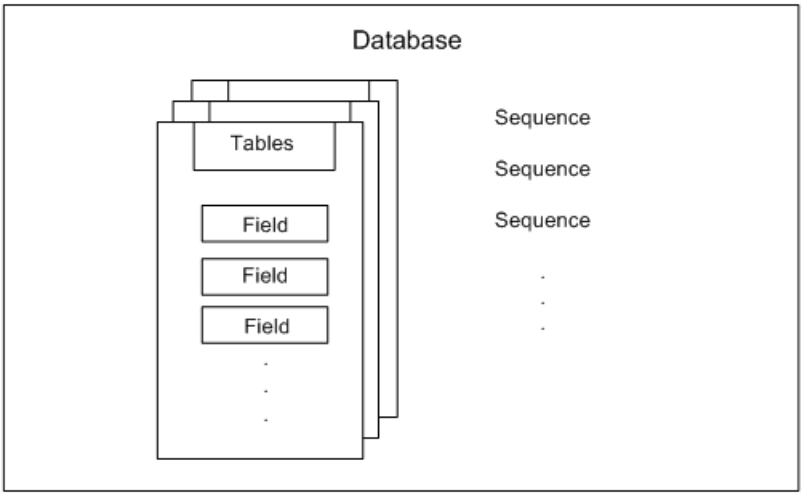
The following sections describe:

- Choosing between sequences and control tables on page 82
- Creating and maintaining sequences on page 85
- Accessing and incrementing sequences on page 85

Choosing between sequences and control tables

Sequences are objects, like fields, that you can create and maintain in any OpenEdge database. Unlike fields, sequences reside in a separate database block independent of application tables. The following figure shows the basic relationship between database fields and sequences.

Figure 7: Database fields and sequences comparison



Each sequence has a name and a value, similar to a field. But unlike a field, a sequence also has different schema attributes that determine how the sequence value can change. These include:

- Whether the sequence increments or decrements and by what interval
- Whether the sequence cycles or terminates
- The upper and lower boundaries for sequence values

ABL provides a dedicated set of functions and statements to access and increment sequences strictly according to the defined attributes.

Using sequences or control tables

Before Progress Version 7, the only way ABL could generate sequential values was to maintain integer fields in a table record created specifically for this purpose. This is often done in a control table, separate from all other application tables.

For example, you might have a control table named syscontrol that contains the field last–cus–num. This field holds the value of the `CustNum` field for the last `Customer` record. The following code fragment generates a new **Customer** record with a unique Customer number:

```
. . .
DEFINE VARIABLE next-custnum NO-UNDO LIKE Customer.CustNum.

DO FOR syscontrol:
  DO TRANSACTION:
    FIND FIRST syscontrol EXCLUSIVE-LOCK.
    next-custnum = syscontrol.last-cus-num + 1.
    syscontrol.last-cus-num = next-custnum.
  END. /* transaction */
  RELEASE syscontrol.
END.

DO TRANSACTION:
  CREATE Customer.
  Customer.CustNum = next-custnum.
  DISPLAY Customer.
  UPDATE Customer.
END. /* transaction */
. . .
```

Note that access to the syscontrol table must be made within a small transaction to avoid lock contention problems.

Sequences provide a built-in means for generating incremental values, but they are not suitable for all applications. In certain situations (described in the following sections), you might need to use a control table, instead.

The following table compares sequences and control tables.

Table 8: Comparison of sequences and control tables

Capability	Sequences	Control tables
Access speed	Fast	Slow
Transaction independent	Yes	No
Guaranteed order	Yes	No
Auto initializing	Yes	No
Auto cycling	Yes	No
Bounds checking	Yes	No
Database limit	250 per 1K database block size	Field limit

Performance versus capabilities

In general, sequences provide a much faster and more automated mechanism to generate sequential values than control tables. Sequences are faster because their values are all stored together in a single, table-independent, database block, and they do not participate in transactions. As a result, the principal limitations of sequences for certain applications include:

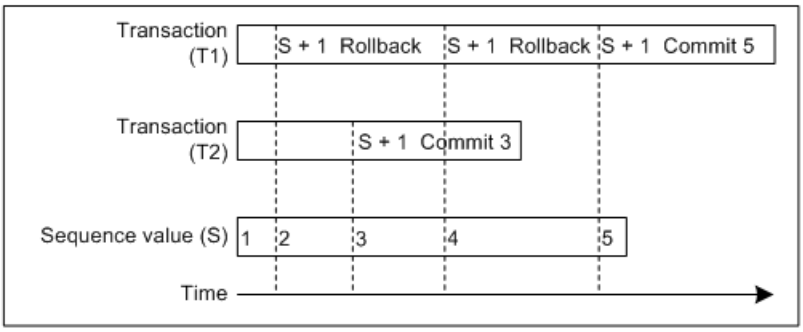
- Because of transaction independence, sequences do not roll back when transactions are undone, but maintain their last value for all UNDOs. However, fields defined in control tables **do** roll back to their pretransaction values, as long as all updates to control tables respect transaction boundaries.
- The number of unique sequences supported by a database varies according to the database block size; whereas the number of fields in a control table is limited by the table size.

Transaction independence

Transaction independence guarantees that each subsequent sequence value is incremented (positively or negatively) beyond its previous value, but does not guarantee the extent of the increment. In other words, with sequences you can have incremental gaps in the sequential values actually used in your application. These gaps result when sequences are incremented during a transaction that is subsequently undone, leaving the sequences set to their latest values. If the transaction restarts, it uses these latest values, not those generated for the previously undone transaction.

The following figure shows how two overlapping transactions with rollback can create records and commit consecutive sequence values with gaps.

Figure 8: Sequences and overlapping transactions



Both transactions start out with the sequence set to 1, which has already been used in the database. Transaction T1 increments the sequence first, assigns, and then rolls back leaving the sequence value at 2. Then transaction T2 increments, assigns, and commits the sequence value 3 to a database field. Transaction T1 then increments, assigns, and rolls back a second time, leaving the sequence value at 4. Finally, transaction T1 increments, assigns, and commits the sequence value 5 to the same database field in a different record. Thus, the sequence values 2 and 4 are skipped and never used in the database. All subsequent transactions increment and commit sequence values greater than 5.

Note: Transaction independence is provided for standard sequence increment/decrement operations, but not for operations that set the sequence value directly. Setting the value directly is a special operation intended only for maintenance purposes. For more information, see Using the CURRENT-VALUE statement on page 88.

Control tables, however, obey the same transaction rules as any other table, and can ensure that their field values are reset for undone transactions. If it is essential to generate sequence values without incremental gaps, you must use a control table rather than a sequence.

Storage limits

The number of sequence objects you can maintain at a time depends on the database block size (250 sequence objects per 1K of database block size). If you need to maintain more sequences, you must either increase the database block size or add another database to define more sequences or us a control table to generate incremental values in your application. The number of incremental fields you can maintain in a control table is limited only by the number of integer fields you can store in the table.

Creating and maintaining sequences

You can create and maintain sequence definitions in a database using OpenEdge's Data Dictionary. OpenEdge stores all sequence values together in a single database block, and stores sequence names and their remaining attributes in a metaschema table named _Sequence.

For information on defining, editing, or deleting sequences in your database, see *Development Tools* (Character only) and, in graphical interfaces, the online Help for the OpenEdge Data Dictionary.

For information on using sequences in ABL, see Accessing and incrementing sequences on page 85.

Providing sequence security

You can restrict sequence access by assigning user privileges to the _Sequence metaschema table. The Can–Read privilege grants users permission to read the current value of a sequence with the CURRENT–VALUE function or to increment and read the next value of the sequence with the NEXT–VALUE function. The Can–Write privilege grants users permission to change the current value of a sequence with the CURRENT–VALUE statement.

Note: Access privileges to the _Sequence metaschema table are in effect during compile-time and during run-time.

For more information on providing security to a database table, see *Manage the OpenEdge Database*.

Dumping and loading sequences

You can dump and load sequence definitions and values using the Database Administration tool. Choose the **Tools** > **Database Administration** from the ADE Desktop. For more information on dumping and loading sequences, see *Manage the OpenEdge Database*.

Accessing and incrementing sequences

The following figure lists ABL statements and functions you can use to access and increment sequence values.

Table 9: Sequence statements and functions

Statement or function	Description
CURRENT–VALUE function	Returns the current value of a sequence identified at compile time.
DYNAMIC–CURRENT–VALUE function	Returns the current value of a sequence identified at run time
NEXT–VALUE function	Increments and returns the incremented value for a sequence identified at compile time
DYNAMIC–NEXT–VALUE function	Increments and returns the incremented value for a sequence identified at run time

Statement or function	Description
CURRENT–VALUE statement	Sets a new current value for a sequence identified at compile time
DYNAMIC–CURRENT–VALUE statement	Sets a new current value for a sequence identified at run time

Whenever a CURRENT–VALUE statement or NEXT–VALUE function changes the value of a sequence, the new value persists in the database where the sequence is defined until it is changed again, or the sequence is deleted from the database.

The basic sequence statements and functions have the following syntax:

Syntax

```
CURRENT-VALUE
  ( sequence-name [, logical-dbname [ , tenant-id ] ] )
 [ = expression ]
```

Syntax

```
NEXT-VALUE
  ( sequence-name [, logical-dbname [ , tenant-id ] ] )
```

sequence-name

An identifier that specifies the name of a sequence defined in the Data Dictionary. Note that a sequence can have the same name as a database field, but they are distinct entities.

logical-dbname

An identifier that specifies the logical name of the database in which the sequence is defined. The database must be connected. You can omit this parameter only if you have a single database connected. Otherwise, you must specify the database in which the sequence is defined.

expression

An expression that evaluates to an integer value. The CURRENT–VALUE statement assigns the value of *expression* to the specified sequence. The value of *expression* must be within the range defined for the specified sequence.

tenant-id

An integer expression that evaluates to a valid tenant ID for a regular tenant of a multi-tenant database. This argument only has meaning for a multi-tenant-enabled sequence. For more information on using multi-tenant sequences, see Using multi-tenant-enabled sequences on page 200.

A dynamic sequence statement and functions have a similar general syntax, but allow you to specify the sequence and database dynamically at run time. Their syntax is as follows:

Syntax

```
DYNAMIC-CURRENT-VALUE
  ( sequence-name-expression , logical-dbname-expression [ , tenant-id ] )
    [ = expression ]
```

Syntax

```
DYNAMIC-NEXT-VALUE
  ( sequence-name-expression, logical-dbname-expression [ , tenant-id ] )
```

sequence-name-expression

A character expression which evaluates to the name of a sequence defined in the Data Dictionary. Note that a sequence can have the same name as a database field, but they are distinct entities.

logical-dbname-expression

A character expression which evaluates to the logical name of the database in which the sequence is defined. The database must be connected. This argument is required and cannot be Unknown value (?).

expression

An expression that evaluates to an integer value. The DYNAMIC–CURRENT–VALUE statement assigns the value of *expression* to the specified sequence. The value of *expression* must be within the range defined for the specified sequence.

tenant-id

An integer expression that evaluates to a valid tenant ID for a regular tenant of a multi-tenant database. This argument only has meaning for a multi-tenant-enabled sequence. For more information on using multi-tenant sequences, see Using multi-tenant-enabled sequences on page 200.

In general, use the CURRENT–VALUE and NEXT–VALUE functions for mission-critical applications that depend on access to reliable and orderly sequence values. Use the CURRENT–VALUE statement only for database maintenance and initialization, primarily during the development cycle. For more information on these functions and statements, see *ABL Reference*.

Using the CURRENT-VALUE function

Use the CURRENT–VALUE function in any integer expression to retrieve the current value of a sequence without incrementing it. The current value of a sequence can be any of the following:

- The Initial value specified in the Dictionary
- The last value set with either the CURRENT–VALUE statement or the NEXT–VALUE function

The following example gets the most recent Customer number in the default database (maintained by the NextCustNum sequence), and displays each order record for that Customer:

```
DEFINE VARIABLE current-cust AS INTEGER NO-UNDO.
. . .
current-cust = CURRENT-VALUE(NextCustNum).

FOR EACH Order NO-LOCK WHERE Order.CustNum = current-cust:
  DISPLAY order.
  . . .
```

Using the NEXT-VALUE function

Use the NEXT-VALUE function to increment a sequence by its defined positive or negative increment value. If the sequence cycles and NEXT-VALUE increments it beyond its Upper or Lower limit, the function sets and returns the defined Initial value for the sequence. If the sequence terminates and NEXT-VALUE tries to increment it beyond its Upper or Lower limit, the function returns the Unknown value (?) and leaves the sequence value unchanged.

The following example creates a new Customer record with the next available Customer number generated by the NextCustNum sequence:

```
DEFINE VARIABLE input-custnum AS INTEGER NO-UNDO.

input-custnum = NEXT-VALUE(NextCustNum, sports2000).
FIND sports2000.Customer NO-LOCK
  WHERE Customer.CustNum = input-custnum NO-ERROR.

DO WHILE (AVAILABLE Customer):
  input-custnum = NEXT-VALUE(NextCustNum, sports2000).
  FIND sports2000.Customer NO-LOCK
    WHERE Customer.CustNum = input-custnum NO-ERROR.
END.

CREATE sports2000.Customer.
Customer.CustNum = input-custnum.
UPDATE sports2000.Customer.
. . .
```

Because this example does not check the NextCustNum sequence for termination, it implies that NextCustNum is a cycling sequence. Because it does check for and ignore existing records containing the generated CustNum value, the example can reuse previously deleted (or otherwise skipped) Customer numbers after the sequence cycles.

Using the CURRENT-VALUE statement

Use the CURRENT-VALUE statement to explicitly set a sequence to a new value. You can assign the defined initial value of a sequence, its upper or lower limit, or any integer value in between. Trying to set a value outside these bounds causes an error. Note that you cannot assign the Unknown value (?) to a sequence. Unlike the NEXT-VALUE function, the CURRENT-VALUE statement always sets a new sequence value in a transaction, starting one, if necessary.

Caution: Avoid using this statement in mission-critical applications. Use of this statement, especially in a multi-user context, can compromise the referential integrity of your database. Any database application designed to rely on the orderly values provided by sequences cannot reliably reset existing database fields according to potentially unexpected sequence values.

The purpose of the CURRENT−VALUE statement is to maintain a database off-line, under program control. Note that the Data Dictionary uses this statement when you create a new sequence to set the initial value. Possible uses include:

- Restarting a terminating sequence that has terminated
- Otherwise, resetting a sequence value for a specific test condition

Keep in mind that all such uses must respect and might require changes to the _Sequence table. This table contains one record for each sequence defined in the database. Creating a record in this table automatically allocates and assigns the data storage for a new sequence. Deleting a record in this table automatically releases the data storage for the specified sequence.

OpenEdge provides fault detection that prevents a sequence from being created with inconsistent attribute values in the _Sequence table (such as, the minimum value greater than the maximum value). It also prevents the assignment of a new current value that is inconsistent with the corresponding attribute values in the _Sequence table (such as, a current value outside the minimum and maximum value boundary).

The following code fragment sets a new current value for sequence used for testing:

```
DEFINE VARIABLE SeqValue AS INTEGER NO-UNDO.

DO TRANSACTION ON ERROR UNDO, RETRY:
  SET SeqValue LABEL "TEST-Stepper Start" WITH SIDE-LABELS.
  CURRENT-VALUE(TEST-Stepper) = SeqValue.
END. /* TRANSACTION */
. . .
```

OpenEdge also uses this statement in the Data Administration tool to load sequence definitions and values from dump-formatted text files. You can accomplish the same task implemented in the code fragment above using this tool. For more information, see *Manage the OpenEdge Database*.

Resetting a terminating sequence

When a terminating sequence stops at its upper or lower limit, you can reset it by assigning a valid sequence value to it with the CURRENT−VALUE statement. You can also reset a stopped terminating sequence by changing it to a cycling sequence. The first use of the NEXT−VALUE function for the new cycling sequence resets the sequence to its initial value.

Referencing sequences within a WHERE clause

You cannot invoke sequence functions from within a WHERE clause. This generates a compiler error, because sequence expressions do not participate in the index resolution and optimization phase of the Compiler.

If you want to use a sequence value within a WHERE clause, set a variable or field to the sequence value, and reference the variable or field in your WHERE clause. See the sections that describe each sequence function for examples.

Database trigger considerations

You can store a database's trigger procedures in an operating system directory or in an ABL r-code library. There are advantages and disadvantages with each technique of storage. Typically, when deploying an application, it is advantageous to store your trigger procedures in an r-code library. When developing the application, it is usually better to store the procedures in a directory.

The Trigger Location (–trig) startup parameter allows you to specify the directory or r-code library containing your trigger procedures.

Storing triggers in an operating system directory

In a development environment, it is usually a good idea to store trigger procedures in a directory. In a directory, you can store compiled procedures (.r files) and source procedures (.p files). You can modify the source code of the procedures as necessary.

Also, you can run uncompiled procedures against any database that has the appropriate schema. In contrast, if a procedure is precompiled, the r-code stores the logical name of the database you compiled the procedure against. The precompiled procedure can only run against a database with the same logical name.

Storing triggers in an r-code library

When deploying your application, it is usually a good idea to place trigger procedures in an r-code library. This is usually faster than running the procedures from an operating system directory, even if the procedures are precompiled. However, procedures run from a library **must** be precompiled. Moreover, because they are precompiled, they are necessarily associated with a logical database name. You can only run them against a database connected with the same logical name.

Using the RAW data type

ABL supports the RAW data type, which lets you manipulate raw data without converting it in any way.

You can use the RAW data type to:

- Define ABL variables in an ABL procedure
- Define fields in a OpenEdge database
- Retrieve and manipulate raw data from non-OpenEdge databases

You can use the RAW data type to import non–OpenEdge data that has no parallel OpenEdge data type. By using the RAW data type statements and functions, ABL allows you to bring data from any field into your procedure, manipulate it, and write it back to the non-OpenEdge database. The functions and statements let you define RAW data type variables, write data into a raw variable, find the INTEGER value of a byte, change the length of a raw variable, and perform logical operations.

The following procedure demonstrates how to retrieve raw values from the database, how to put bytes into variables, and how to write raw values back to the database:

```
/*You must run this procedure against a non-OpenEdge sports2000 database.*/

DEFINE VARIABLE r1 AS RAW     NO-UNDO.
DEFINE VARIABLE ix AS INTEGER NO-UNDO.

FIND FIRST customer.
r1 = RAW(Customer.Name).
PUT-BYTE(r1,1) = 115.
RAW(Customer.Name) = r1.
DISPLAY Customer.Name.
```

This procedure first creates the variable r1 and defines it as a RAW data type. Next, it finds the first Customer in the database ("Lift Line Skiing"), and with the RAW function, takes the raw value of the Name field, and writes it into the variable r1. The PUT–BYTE statement then puts the character code value of "s" (115) into the first byte of r1. The RAW statement takes the raw value of r1 and writes it back to the database. Finally, the procedure displays the Customer name. Thus, "Lift Line Skiing" has become "sift Line Skiing".

The next procedure shows how you can pull bytes from a field:

```
/*You must run this procedure against a non-OpenEdge sports2000 database.*/

DEFINE VARIABLE ix AS INTEGER NO-UNDO INITIAL 1.
DEFINE VARIABLE ax AS INTEGER NO-UNDO.

FIND Customer WHERE Customer.CustNum = 27.

REPEAT:
  ax = GET-BYTE(RAW(Customer.Name),i).
  DISPLAY ax.
  IF ax = ? THEN LEAVE.
  ix = ix + 1.
END.
```

This procedure finds the `Customer` with the `Customer` number 27, and then finds the character code value of each letter in the `Customer` name. To do this, it retrieves the bytes from the name one at a time, then places them into the variable a. The `GET-BYTE` function returns the `Unknown value (?)` if the byte number you try to retrieve is greater than the length of the expression from which you are retrieving it. The next procedure demonstrates how you find the length of a raw value and how to change length of a raw expression:

```
/* You must run this procedure against a non-OpenEdge sports2000 database. */
DEFINE VARIABLE r3 AS RAW.

FIND FIRST Customer.
r3 = RAW(Customer.Name).
DISPLAY LENGTH(r3) Customer.Name WITH DOWN. /* length before change */
DOWN.

LENGTH(r3) = 2.
DISPLAY LENGTH(r3) Customer.Name.           /* length after change */
```

This procedure simply finds the number of bytes in the name of the first `Customer` in the database then truncates the number of bytes to two. The procedure first displays 16 because the `Customer` name Lift Line Skiing contains 16 bytes. It then displays 2 because you truncated the raw value to two bytes.

Note: When you use the `STRING` function to retrieve a raw value as a string, you must supply a format for the string as the second function argument.

Multi-database programming techniques

Once you've settled on a design and run-time connection technique for your multi-database application, you can begin programming. The following sections list a variety of programming techniques and issues to consider as you develop your multi-database application with OpenEdge.

Referencing tables and fields

Unique table and field names do not require fully qualified references within application procedures. In a single database OpenEdge application, references to non-unique field names require a table prefix to avoid ambiguous field references. Use the following syntax to reference a non-unique field in a procedure:

Syntax

```
table-name.field-name
```

The *table–name* is the name of the database table that contains the field *field–name*.

The ability to connect to several databases from a single OpenEdge session introduces the possibility of non-unique table names and increases the possibility of non-unique field names. References to non-unique table names within multi-database OpenEdge applications require database prefixes to avoid ambiguous table references. Ambiguous table and field references cause compilation failures.

Use the following syntax to include a database prefix in a table or field reference:

Syntax

```
database-name.table-name
database-name.table-name.field-name
```

The *database–name* is the logical name (or an alias for a logical name) that represents the database that contains the table *table–name*.

For example, suppose you are connected to two databases db1 and db2, both of which contain a table called `Customer`. The following procedure will not compile due to an ambiguous table reference:

```
FOR EACH Customer: /* In db1 or db2 ? */
  DISPLAY Customer.Name.
END.
```

The procedure below uses fully qualified table references to display Customer names from both connected databases (db1 and db2):

```
FOR EACH db1.Customer:
  DISPLAY Customer.Name.
END.
FOR EACH db2.Customer:
  DISPLAY Customer.Name.
END.
```

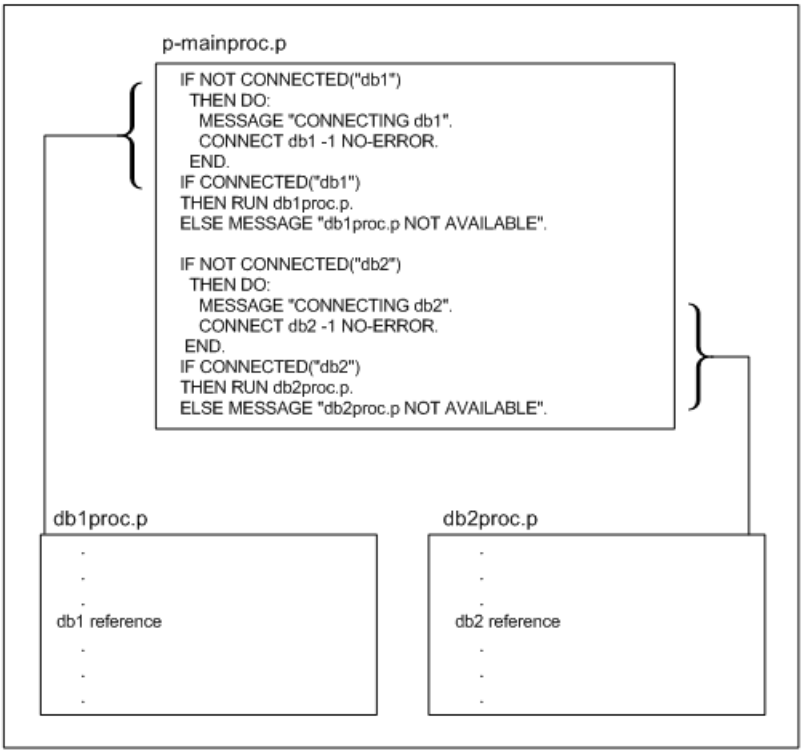
Notice that the two references to the name field do not need to be qualified because they appear within a `FOR EACH` block that already contain fully qualified table references.

Note: When the AVM encounters a statement such as `DISPLAY X.Y`, it first attempts to process `X.Y` as a *databasename.table–name*. If that fails, the AVM then attempts to process `X.Y` as *table–name.field–name.*

Positioning database references in an application

If possible, isolate database references within your application to help minimize the effects of a database connection failure on your application. This allows you to use the `CONNECTED` function effectively to test for a particular database connection, prior to passing program control to a subprocedure that accesses that database.

Figure 9: Positioning database references



The above figure shows a main procedure that connects two databases and runs subprocedures, depending on whether or not a database is connected. If the `CONNECT` statement in `mainproc.p` fails to connect db2, only `db2proc.p` subprocedure is effected by the database connection failure. This technique is useful for applications that run subprocedures from a menu.

Do not reference a database in your application startup procedure, except possibly to connect it. Remember, you cannot connect to a database and reference a database in the same procedure. If you reference a database on an auto-connect list within the startup procedure of your application and the auto-connect fails for that database, the startup procedure does not run.

Using the LIKE option

The ABL `LIKE` option in a `DEFINE VARIABLE` statement, `DEFINE WORKFILE` statement, `DEFINE WORK–TABLE`, or Format phrase requires that a database is connected. Use the `LIKE` option with caution in procedures that do not otherwise access a database.

Creating schema cache files

When OpenEdge starts a client application, it loads the schema for each database that the application requires into memory. Ordinarily, OpenEdge reads the schema from the database, which can be on a remote server. If the network is slow, or the server, itself, is overloaded, this can increase startup time significantly.

You can shorten client startup time by building and saving a schema cache file on a disk local to the client. A *schema cache file* is a machine-portable binary file containing a complete schema or subschema for a single OpenEdge database. When you start a client application with local schema cache files, the AVM reads the required schema for each database almost instantaneously from the local disk rather than waiting for network or server traffic.

Building a schema cache file

To build a schema cache file, you use the SAVE CACHE statement together with a connected database. You must first decide whether you need the entire schema cache or only the schema cache for selected tables of a database. If your application accesses all the tables in the database, you need the complete cache. Otherwise, you can build a schema cache file for only the tables that are accessed by your application.

In general, you build a schema cache file off-line, after making a schema change in the database. You can do this in the Procedure Editor directly, or you can write a small maintenance procedure to generate the file.

This is the syntax of the SAVE CACHE statement:

Syntax

```
SAVE CACHE { CURRENT | COMPLETE }
  database-name TO pathname
```

The *database–name* can be the literal logical name of any OpenEdge database or the VALUE(*expression*) option, where *expression* is a character expression that evaluates to the database name.

Note: For a DataServer, OpenEdge saves the schema cache for the entire schema holder database. You cannot save the schema cache for a non-OpenEdge database separately. For more information on schema cache files for DataServers, see your OpenEdge DataServer guide.

The *pathname* can be the literal pathname of an operating system file or the VALUE(*expression*) option, where *expression* is a character expression that evaluates to the pathname.

To save the entire schema cache:

1. Connect to the database.
2. Execute the SAVE CACHE statement using the COMPLETE option.

For an example procedure that saves the entire schema cache for one or more databases, see the SAVE CACHE Statement reference entry in *ABL Reference*.

To save a partial schema cache for selected tables of a database:

1. Connect to the database.
2. Read each table you want to include in the schema cache with a FIND statement.
3. Execute the SAVE CACHE statement using the CURRENT option.
4. If you want to save a different partial schema cache for the same database, disconnect the database and repeat Steps 1 through 3.

Example schema cache file creation

The following event-driven application allows you to create partial schema cache files for any combination of tables in the sports2000 database.

The main procedure, `i-schcs1.p`, presents all the available `sports2000` database tables in a multiple selection list and a field to enter a name for the schema cache file.

i-schcs1.p

```
DEFINE VARIABLE filen AS CHARACTER NO-UNDO FORMAT "x(8)"
  LABEL "Schema Cache Name".
DEFINE VARIABLE icnt  AS INTEGER   NO-UNDO.

DEFINE VARIABLE db-table AS CHARACTER LABEL "Select Tables"
  VIEW-AS SELECTION-LIST
  MULTIPLE NO-DRAG SIZE 32 BY 7
  LIST-ITEMS "customer", "invoice", "item", "localdefault",
             "order", "orderline", "refcall", "salesrep",
             "state".

DEFINE BUTTON bsave   LABEL "Save to File".
DEFINE BUTTON bcancel LABEL "Cancel".

DEFINE FRAME SchemaFrame
  SPACE(1)
  db-table
  VALIDATE(db-table <> "" AND db-table <> ?, "You must select a table.")
  filen
  VALIDATE(filen <> "" AND filen <> ?, "You must enter filename.")
  SKIP(1)
  SPACE(20) bsave bcancel
  WITH TITLE "Save Schema Cache File" SIDE-LABELS SIZE 80 by 11.

ON CHOOSE OF bcancel IN FRAME SchemaFrame QUIT.

ON CHOOSE OF bsave IN FRAME SchemaFrame DO:
  ASSIGN filen db-table.
  IF NOT filen:VALIDATE() THEN RETURN NO-APPLY.
  IF NOT db-table:VALIDATE() THEN RETURN NO-APPLY.
  DO WHILE NOT CONNECTED("sports2000"):
    BELL.
    PAUSE MESSAGE
      "When ready to connect the sports2000 database, press <RETURN>".
    CONNECT sports2000 -1 NO-ERROR.
    IF NOT CONNECTED("sports2000") THEN
      DO icnt = 1 to ERROR-STATUS:NUM-MESSAGES:
        MESSAGE ERROR-STATUS:GET-MESSAGE(icnt).
      END.
    ELSE
      MESSAGE "Sports2000 database connected.".
  END.
  RUN i-schcs2.p (INPUT db-table, INPUT filen).
  DISCONNECT sports2000 NO-ERROR.
END. /* ON CHOOSE OF bsave */

ENABLE ALL WITH FRAME SchemaFrame.
WAIT-FOR CHOOSE OF bcancel IN FRAME SchemaFrame.
```

When you click **Save to File**, it connects to the `sports2000` database, and calls `i-schcs2.p`, which reads the selected tables and saves the resulting schema cache file in the current working directory.

i-schcs2.p

```
DEFINE INPUT PARAMETER db-table AS CHARACTER NO-UNDO.
DEFINE INPUT PARAMETER filen    AS CHARACTER NO-UNDO.

DEFINE VARIABLE iTab AS INTEGER NO-UNDO.

Table-Add:
DO iTab = 1 to NUM-ENTRIES(db-table):
  CASE ENTRY(iTab, db-table):
    WHEN "customer"     THEN FIND FIRST Customer NO-ERROR.
    WHEN "invoice"      THEN FIND FIRST Invoice NO-ERROR.
    WHEN "item"         THEN FIND FIRST Item dO-ERROR.
    WHEN "localdefault" THEN FIND FIRST LocalDefault NO-ERROR.
    WHEN "order"        THEN FIND FIRST Order NO-ERROR.
    WHEN "orderline"    THEN FIND FIRST OrderLine NO-ERROR.
    WHEN "refcall"      THEN FIND FIRST RefCall NO-ERROR.
    WHEN "salesrep"     THEN FIND FIRST SalesRep NO-ERROR.
    WHEN "state"        THEN FIND FIRST State NO-ERROR.
    OTHERWISE           LEAVE Table-Add.
  END CASE.
END.

SAVE CACHE CURRENT sports2000 TO VALUE(filen + ".csh") NO-ERROR.
IF NOT ERROR-STATUS:ERROR THEN
  MESSAGE "Saved partial schema cache for the sports2000 database in "
    + filen + ".csh.".
ELSE DO:
  BELL.
  DO iTab = 1 TO ERROR-STATUS:NUM-MESSAGES:
    MESSAGE ERROR-STATUS:GET-MESSAGE(iTab) VIEW-AS ALERT-BOX.
  END.
END.
```

The main procedure then disconnects the `sports2000` database to allow new table selections for a different schema cache file.

Using a schema cache file

You specify the schema cache file for each database of an application using the Schema Cache File (–cache) startup parameter. If the schema cache file for any database is invalid, the AVM displays a message, ignores the file with a warning, and reads the cache from the database. For example:

```
The time stamp in the database does not match the time stamp in the cache
  file: sports2000.csh (840)
```

The user can notify the database administrator or application developer to provide the correct schema cache file. For more information on using schema cache files, see *Manage ABL Applications*, and for a description of `–cache`, see *Startup Command and Parameter Reference*.

2

Application Security

As a developer, you can provide your ABL applications with security features that prevent unauthorized users from running application procedures or accessing data.

For details, see the following topics:

- Types of application security
- Elements of an application authentication model
- Elements for authenticating ABL user identity
- Authenticating and managing user identity in ABL
- Authorizing access to procedures and database resources
- Using ABL callbacks in authentication systems
- Using cryptography to secure data

Types of application security

OpenEdge provides the following types of security for an ABL application:

- **Identity management** — Primarily supports the following security services:
  - **Authentication** — Ensures that only a user with valid user credentials can login to and run an application or connect to a particular database. Once validated, authentication seals the user credentials in a portable security token. The AVM uses the sealed security token to authorize user access to ABL sessions, applications or database connections within a domain.

  - **Authorization** — Ensures that only authorized users can read or change application and database options and resources, such as data tables and fields, and the table, field, and index definitions (*schema*) in a database. OpenEdge supports *compile-time authorization*, which ensures that only authorized users can compile procedures and classes that access specific database tables and fields, and *run-time authorization*, which ensures that only authorized users can access specific database tables and fields or run specific precompiled procedures and classes.

- **Cryptography** — Ensures data privacy and integrity so that unauthorized users cannot read or change data, whether it is stored in a database, stored in an ABL session context, or transported over a network.
- **Auditing** — A means of securely tracking and recording a trail of events that occur during execution of an application. The basic security in auditing allows this trail of events (audit events) to be recorded in a manner that cannot be changed or otherwise repudiated after the fact. You can use ABL to record application-defined audit events and also to build custom audit configuration tools.

These topics primarily describe how to use the ABL features that support identity management and cryptography. For a comprehensive overview of OpenEdge identity management, including ABL support, see *Introduction to Identity Management*. For a general introduction to OpenEdge security, cryptography, and a comprehensive guide to OpenEdge auditing, see *Introduction to Security and Auditing*. For information making secure network connections between ABL clients and database servers, see the information on database connections in Database Access on page 35. For information on creating secure sockets in ABL, see Sockets on page 579. For more information on ABL support for auditing, see Auditing on page 241

The security administrator can define security options for database connections, schema changes, and compile-time authorization in the Data Dictionary and in other database administration tools to manage secure database connections. OpenEdge also uses the same settings for compile-time authorization in order to handle run-time authorization for database table and field access. For more information on database security administration, see *Manage the OpenEdge Database* and *Manage ABL Applications*.

As described in the rest of these topics, the developer is usually responsible for authenticating users, managing user identity, connecting securely to a database, authorizing access to application resources, and managing cryptographically secured data within an ABL session.

Elements of an application authentication model

In OpenEdge, you can implement any one of a variety of authentication models. The *authentication model* defines and determines how (and if) you use any of the authentication elements described in this section.

For more information on the basic features provided by OpenEdge to include in an application authentication model, see *Introduction to Identity Management*. Other sections describe how to use the supported mechanisms to implement these authentication features for an ABL application.

Authentication systems

An OpenEdge *authentication system* ensures that user credentials are valid. If the credentials are valid, the authentication system creates a sealed security token that represents the user identity within a security domain. The AVM uses the sealed security token to authorize user access to ABL sessions, applications or database connections within that domain. The sealed security token remains available after initial authentication. This enables single sign-on (SSO), where the AVM uses the sealed security token for subsequent authentication rather than requiring the user to login again.

OpenEdge includes several built-in authentication systems, namely `_oeusertable`, `_oslocal`, and `_extsso`. You can extend the functionality of the built-in authentication systems by using ABL callbacks. In addition, you can use ABL callbacks to implement authentication in a security domain that you define.

See Authentication systems supported in ABL on page 101 for more information on built-in and external authentication systems. Also see Specifying user IDs and passwords in ABL on page 102 for information on callback procedures.

Security domains

In OpenEdge, the authentication system used to authenticate a given user identity is identified by an *OpenEdge security domain* (or *domain*). The OpenEdge domain identifies the authentication system and provides a *domain access code* to validate the identity of all users who are authenticated in a particular domain.

A particular domain also allows all authenticated users to share other common attributes, such as a common tenancy in a multi-tenant database and (through domain authorization settings) access to a common set of database and application resources.

In OpenEdge, a security administrator defines and configures domains in each OpenEdge RDBMS and enables or disables each domain for run-time use. A domain is *authentication-enabled* if it is enabled for run-time use, and is configured with an authentication system that has access to valid user accounts and supports user authentication. For more information on OpenEdge support for domains and authentication systems, see Authentication systems supported in ABL on page 101.

Domain registries

At run time, OpenEdge provides identity-management access to security domains through a *domain registry*. OpenEdge provides:

- A default *local database domain registry* to access domains to authenticate OpenEdge database connection identities.
- ABL elements for creating an *ABL session domain registry* to access domains to authenticate a single ABL session identity. You also have the option to use the ABL session domain registry to authenticate OpenEdge database connection identities.

The domain registry, or registries, that your application uses depend on security administrator settings for each database and whether (and how) your application creates and uses ABL session registries. In any case, a session identity is always authenticated through the current session domain registry. However, the registry configured to authenticate the identity for a given database connection is referred to as the database *trusted domain registry*. For more information on using domain registries, see Setting up and using domain registries on page 123.

User credentials

User credentials are data elements that identify a user to your application. The user identifies themselves by *asserting* their user credentials for authentication against a valid user account.

The data elements required for user credentials depend on the authentication system and the features you enable in an OpenEdge database. At a minimum, these are the data elements required to initialize a security token that represents the user's identity. For more information on working with user credentials, see Specifying user IDs and passwords in ABL on page 102.

Security tokens (client-principal objects)

A *security token* is an object that encapsulates a single user identity. In ABL, a security token is implemented as an instance of a handle-based client-principal object, which you can use with various statements, functions, and methods to manage a user identity.

You can initialize and assert the identity in an *unsealed* client-principal for *user authentication*. When OpenEdge performs a successful authentication, it then *seals* the client-principal. Once sealed, you cannot change the user identity (or any other settings) in a client-principal, but can use the *sealed* client-principal in a *single sign-on* (*SSO*) operation to validate and assign the identity it represents to any session or database connection in an ABL application. Thus, your application can use a client-principal to both authenticate a user identity for one application resource and assign that identity to other ABL sessions and database connections in the application.

Progress Software **strongly** recommends that you use client-principal objects as the basis of any authentication model that you design. This object provides full support for all authentication and identity management features in ABL and offers a security mechanism that OpenEdge supports from release to release. In addition, you **must** use client-principal objects to authenticate and assign the connection identity for an OpenEdge multi-tenant database.

For more information on client-principal objects, see Security tokens in ABL on page 103.

User identity management mechanisms

ABL supports several mechanisms to authenticate and set a given user identity for one or more database connections and ABL sessions.

Some mechanisms (the `SET-DB-CLIENT` function and `SET-CLIENT( )` method on the `SECURITY-POLICY` system handle) take an unsealed client-principal object that you can create and initialize with user credentials in order for OpenEdge to authenticate the user identity. Then, using the same mechanisms in an SSO operation with the sealed client-principal, you might later authenticate the identity to other existing database connections and ABL sessions. These mechanisms are the only way to set the identity for a multi-tenant database connection, and the `SET-CLIENT( )` method is the only mechanism available to set an ABL session identity.

Other mechanisms (like the User ID (`-U`)/Password (`-P`) connection parameters and the `SETUSERID` function) take user credentials directly as input for OpenEdge to authenticate and set the identity for new or existing database connections. Successful authentication using these direct authentication mechanisms automatically creates a sealed client-principal object that you can return to set the same identity for additional resources (ABL sessions and database connections) in your application.

For more information on authenticating and setting user identities, see Authenticating and setting user identity on page 128.

Client context management mechanisms

Client context management mechanisms identify and maintain context for a single user across all ABL sessions of a multi-tier application running in session-free or session-managed operating mode.

ABL supports different mechanisms to do this that work depending on the application session model (session-free or session-managed). This includes a *client context identifier* (ID) that you can transport, along with a client-principal object, across the sessions of an application server application in order to identify and restore the user's current context, regardless of the application session model.

For more information using this client context ID mechanism, see Establishing and managing identity for multi-tier applications on page 145.

Application authentication design options

If your application performs authentication, there are three basic design options for using a client-principal object:

- **Client authentication** — The client session authenticates its own user identity and seals the client-principal object (starting a user login session). Client authentication is most appropriate for a client-server application with limited or no access to an application server.

- **Server authentication** — An unsealed client-principal object is initialized with user credentials in the user's client session, then exported to an *authentication server* (application server), which authenticates the identity, seals the object (starting a user login session), and returns the sealed client-principal back to the user's client (and to any other application server session in the application on request) in order to establish session and database connection identity using SSO. The authentication server typically also manages the logout, termination, and clean up of the login session for the user. Sever authentication is most appropriate for a multi-tier application, but is potentially the most secure authentication option for any ABL application, especially if it resides on a locked-down server where all server code is protected from access by anyone except a local server administrator.

Elements for authenticating ABL user identity

As noted in previous sections, *authentication* is the process of validating a user's identity. The identity validation mechanism depends on the authentication system used, and whether it supports user authentication, single sign-on (SSO), or both. Generally, if an authentication system supports user authentication, it also supports SSO. If it supports only SSO, the ABL application must perform the user authentication. For more information on OpenEdge support for user authentication, SSO, and authentication systems, see *Introduction to Identity Management*.

The following sections describe the basic features for authenticating user identities in ABL applications. For more information on managing user identities in ABL, see Authenticating and managing user identity in ABL on page 104

- Authentication systems supported in ABL on page 101
- Specifying user IDs and passwords in ABL on page 102
- Password encryption on page 103
- Security tokens in ABL on page 103

Authentication systems supported in ABL

For ABL applications, OpenEdge supports for the following types of authentication systems:

- **OpenEdge internal (`_oeusertable` system)** — OpenEdge performs the user authentication to the OpenEdge `_User` table accounts established in an OpenEdge RDBMS.

  Although built-in, this authentication system can be configured with an ABL callback procedure that customizes the criteria for authentication, for example, limiting the number of times a user can login per day. This authentication system is supported for both OpenEdge-performed user authentication and SSO.

- **Local operating system (`_oslocal` system)** — OpenEdge performs the user authentication to the local operating system user accounts (Windows or Unix).

  Although built-in, this authentication system can be configured with an ABL callback procedure that customizes the criteria for authentication, for example, limiting the number of times a user can login per day. This authentication system is supported for both OpenEdge-performed user authentication and SSO.

- **ABL application** — An ABL application performs the user authentication using either application-managed user accounts or an external user account system.

  To support application-performed authentication, OpenEdge supports one built-in authentication system (`_extsso`) and the configuration of additional user-defined authentication systems. The ABL application must perform the user authentication (including creation of the sealed client-principal object) for both `_extsso` and user-defined authentication systems.These authentication systems are restricted to OpenEdge-performed SSO.

- **ABL callback** — OpenEdge performs the user authentication through a user-defined authentication system implemented by an ABL callback that is independent of any application. OpenEdge runs the callback in the background. OpenEdge also seals the client-principal object when you use an ABL callback (unlike application-performed authentication where the application must seal the client-principal object).

Note: All authentication systems can be configured with an ABL callback that executes after an authenticated user identity is set. For example, the callback might initialize a user context store for use by other ABL sessions in a multi-tier application. For more information, see Using ABL callbacks in authentication systems on page 166.

To authenticate a user identity, an authentication system must be enabled for user authentication, have access to a valid set of user accounts, and be part of an OpenEdge security domain configuration that is also run-time enabled. With this configuration, the domain is then authentication-enabled. To use an authentication-enabled domain for authentication, you must include the domain name with the user name as the fully qualified user ID in an OpenEdge authentication operation

However, you can implement an authentication system even when a domain is not authentication-enabled. For domains that are not authentication-enabled, your application must implement the user authentication itself, either by managing its own store of users or by accessing an external source of user accounts such as LDAP. For more information on differences between OpenEdge and application-performed user authentication, see Authenticating and setting user identity on page 128.

Note: OpenEdge supports a default blank domain for a user that does not otherwise specify a domain.

ABL also supports several mechanisms to validate and assign a user identity through SSO, which works with all authentication systems. Regardless of the mechanism used, OpenEdge performs the actual SSO operation that validates the user identity.

For more information on OpenEdge support for, and configuration of user IDs, authentication systems, and security domains, see *Introduction to Identity Management*.

Specifying user IDs and passwords in ABL

The requirements for user IDs and passwords vary according to the user account system that defines them. For more information on forming user IDs and passwords, see *Introduction to Identity Management*. For information on defining user IDs and passwords for the database user list, see the sections on maintaining security in *Manage the OpenEdge Database*.

Note that when you add a password for users defined in the database user list, the password is encoded with the ABL `ENCODE` function. **All** OpenEdge **internal passwords are case sensitive.** So, `ENCODE` generates different values for uppercase and lowercase input.

You might also use this function to store passwords for any user-defined authentication system that you implement. When using `ENCODE` with passwords, Progress Software Corporation strongly recommends that:

- Invocations of the `ENCODE` function for the same user run in the same code page.
- In environments with multiple code pages, programs use the `CODEPAGE-CONVERT` function so that invocations of the `ENCODE` function for the same user run in the same code page.

The `ENCODE` function performs a one-way encoding operation that you cannot reverse. It is useful for creating scrambled copies of clear-text password strings that can be stored in a database. However, it is not possible to decode the string returned from the `ENCODE` function. You cannot use the `ENCODE` function to generate an encrypted password that can be used in an ABL security token or with an OpenEdge startup parameter.

Note: The ENCODE function is not a strong encoding operation and should be combined with a salt value for greater security. See Using cryptography to secure data on page 175 for more information about creating and using salt values.

Beyond the requirements of individual user account systems, the maximum character length of user IDs is limited to the maximum length of the ABL CHARACTER data type. For more information, see the data types section in *ABL Reference*.

Password encryption

You can protect password information during authentication using ABL password encryption. To programmatically generate an encrypted password:

1. Use the ENCRYPT-AUDIT-MAC-KEY( ) method of the AUDIT-POLICY system handle to generate an encrypted version of a clear-text password.
2. Add the encryption format prefix, oech1, to the encrypted password using the double colon (::) separator.

   The encryption format prefix, oech1, specifies the encryption algorithm (oec), the encoded password format (h), and the encryption keypad (1). The prefix is necessary so that OpenEdge will properly interpret the string as an encrypted value. Without the prefix, the string would be processed as if it were clear-text.
3. Assign the encrypted password string to a CHARACTER variable.

For example:

```
DEFINE VARIABLE encpwd AS CHARACTER NO-UNDO.
encpwd = "oech1::" + AUDIT-POLICY:ENCRYPT-AUDIT-MAC-KEY("clear_text")
```

The encrypted password, thus generated, can be substituted for a clear-text value in the following contexts:

- As an option in a CONNECT *db-name* -U *user-id* -P *encpwd*statement
- As a password or domain access code parameter for the client-principal methods: INITIALIZE( ), SEAL( ), and VALIDATE-SEAL( )
- As a value for the client-principal PRIMARY-PASSPHRASE attribute
- As a password parameter for the SETUSERID( ) function

Note: You can also generate an encrypted password using the genpassword utility. This method is useful when you need an encrypted password as a startup parameter value on the command line or in a .pf file. For more information, see the encrypted password procedure in the Password (-P) reference entry of the *Startup Command and Parameter Reference*.

Security tokens in ABL

The product of all user authentication operations in ABL is a sealed security token that securely encapsulates the user's identity and establishes that identity for connection to an OpenEdge RDBMS, for access to application resources in an ABL session, or for access to an entire ABL application, including all the connected ABL sessions and databases of a multi-tier application. Thus, a single security token allows an ABL application to transport a user identity from one part of an application to another, whether for initial user authentication or to establish the user's identity for access to a possibly different database connection or ABL session.

In ABL, a security token is implemented as a *client-principal* object. This is a handle-based object that supports the mechanisms for authenticating a user identity, whether it is through user authentication or SSO. Depending on the context, either OpenEdge or the ABL application instantiates and seals a client-principal object for each authenticated user identity. ABL provides a number of built-in functions, statements, and handle methods that either consume or produce a client-principal object to manage a given user identity. For more information on these ABL elements, see ABL for managing user identity on page 106.

For more information on features, methods, and attributes of client-principal objects, see Client-principal objects on page 111.

Authenticating and managing user identity in ABL

OpenEdge supports two basic types of user identity in ABL:

- **Authenticated user identity** — An identity that is authenticated against a trusted domain registry and sealed in a client-principal object, either by OpenEdge or by an ABL application. The user ID for an authenticated identity can have any valid character-string value, depending on the authentication system and OpenEdge domain, except that the account user name cannot contain the `'@'` character, which is used to separate the account user name from the OpenEdge domain name in a fully qualified user ID. However, an authenticated user ID can include blank (`""`), which means a blank account user name and a blank OpenEdge domain name, if the authentication system supports it (as in the database user list). Once authenticated, this identity can be validated in the sealed client-principal using an SSO operation to set the identity for any ABL session or database connection that allows it.
- **Default user identity** — Supported for backward compatibility, an identity that OpenEdge assigns to a database connection that is established **without** using the User ID (`-U`) and Password (`-P`) connection parameters. If an authentication-enabled domain exists in the database, OpenEdge assigns blank (`""`) as the default user ID; otherwise, OpenEdge assigns the user ID of the local operating system process making the database connection. You **cannot** invoke an SSO operation on a client-principal object created and sealed by OpenEdge for the default user identity. In other words, the default identity can **only** apply to the database connection on which it is established, and cannot be assigned using any subsequent identity management operation.

Note: You can configure how the database handles blank user IDs for both authentication to a database connection and authorization for table and field access. The **Security** menu of the Data Administration tool provides an option to disallow blank user ID access to tables and fields that ABL code accesses at both compile-time and run-time. This menu also provides a set of additional database options, including an option to disallow connection to the database using the blank user ID. These restrictions on use of the blank user ID apply to both the OpenEdge default or an authenticated blank user ID. For more information, see the sections on security in *Manage ABL Applications* and the Data Administration tool online help.

For more information on OpenEdge user identity types and how OpenEdge determines the user ID for the default user identity, see *Introduction to Identity Management*.

The following sections describe how to establish and manage ABL user identity for various identity applications.

- User identity applications on page 105
- ABL for managing user identity on page 106
- Client-principal objects on page 111
- Setting up and using domain registries on page 123
- Authenticating and setting user identity on page 128
- Establishing database connection identity on page 135

- Establishing ABL session identity on page 141
- Establishing and managing identity for multi-tier applications on page 145
- Establishing auditing identity on page 154
- Creating and managing unique object identities on page 154

User identity applications

OpenEdge supports management features for one or more of the following user identity applications, depending on configured database options:

- **Database connection identity** — A *database connection identity* is the user identity assigned to a connected OpenEdge RDBMS. This can be an authenticated identity or an OpenEdge default identity. To ensure that a database is connected with an authenticated identity, you can establish the connection with the `CONNECT` statement or use the `SET-DB-CLIENT` function on an existing connection to set an authenticated identity. Initially, database access is fully open to all users. Using database options set by the security administrator, OpenEdge can then authorize access to specific database tables and fields or to specific database management functions (for security administrators, for example) based on the authenticated or default connection identity. The ABL `USERID` function returns the user ID of the current database connection identity, regardless of how it is set (by authentication or default).
- **ABL session identity** — An *ABL session identity* is a user identity assigned to a given ABL session, independent of any database connections. To establish a session identity, you must invoke the `SET-CLIENT( )` method on the `SECURITY-POLICY` system handle; otherwise, the session identity is undefined. The ABL session identity can thus be used to authorize user access to application features in a database-independent fashion. These can be features that are entirely application defined or that are supported specifically by OpenEdge, such as the auditing identity (see the next bullet). Depending on database options you set and how your application handles the authentication of database connections, setting the ABL session identity can also set the identity for all database connections in the session.
- **Application user identity** — An *application user identity* is a common user identity established by a multi-tier application for use by all ABL sessions that run on behalf of the application. Typically, the application user identity is shared between a single OpenEdge application server client and the application server session(s) that process client requests. Depending on the application session model, this single application user identity can also be shared between a single OpenEdge client session and multiple application server instances. Each application server session uses the application user identity to set the ABL session identity and any or all database connection identities required by the session. You typically set the application user identity from a user ID that is asserted to the client session and authenticated by an application server running as an authentication service.
- **Auditing identity** — An *auditing identity* is the designated user identity that OpenEdge records in the audit event records of an audit trail. There is no functionally independent auditing identity. Instead, the auditing identity is set from one of the other established identities, depending on the application configuration. By default, the auditing identity for the audit trail recorded by a given database is the database connection identity for that database. However, you can also set a database option to use the identity of any ABL session that connects to the database as the auditing identity. In this way, you can configure auditing for every database that is connected to a given ABL session so that all database audit trails for that session are associated with the same user identity. For more information on auditing identity, see *Introduction to Security and Auditing*.

In any given ABL application, you might use none, some, or all of these user identities to secure and identify features and data. Also, the management of user identities can be more or less complex depending on whether your application is client-server or multi-tier. ABL provides features to handle both.

ABL also provides features for creating and managing a unique identity for client context. You might use this client context ID as part of a current multi-tier application user identity to key database objects you might maintain for storing and retrieving client context across all the sessions that execute on behalf of the user.

ABL for managing user identity

ABL for managing user identities consists of the elements shown in the following table.

Table 10: ABL elements for managing user identities

This ABL element...	Provides this identity management function...
Buffer object handle and buffer-field object handle	Provides `LOGICAL CAN-*` attributes for testing run-time data access permissions for the current database connection identity. If the current database user identity has permission to access a given database table or field using the data (CRUD) operation associated with a specific `CAN-*` attribute, the attribute returns `TRUE`. For more information, see Authorizing user access to tables and fields on page 165.
`CAN-DO` function	Matches a specified user ID against an ID pattern list, which you can use to authorize access to application data and features. For more information, see Authorizing access to procedures and database resources on page 155.

This ABL element...	Provides this identity management function...
Client-principal object handle	When unsealed, the referenced object can be used to start a user login session for a single user identity after authenticating and sealing the user identity in the object. When sealed, the object, and copies of the object, can be used to maintain a user login session for the user identity it represents across multiple database connections or the ABL sessions of a multi-tier application. A client principal object contains a variety of information about a login session and the identity it represents, including its state and information necessary to validate and set the user identity for an ABL session or database connection using a trusted domain registry.[2], [3] Using its attributes and methods, you can initialize an unsealed client principal for user authentication, transport a sealed or unsealed object between ABL sessions, and change the state of the object in various ways. For more information, see Client-principal objects on page 111.

[2] Each connected OpenEdge RDBMS has a local domain registry that contains information about each domain in which a user identity can be authenticated or validated to set the database connection identity. You can optionally configure the database, instead, to use the application (session) domain registry of the ABL session to which the database is connected to provide information about these domains. For more information, see Setting up and using domain registries on page 123.

[3] Each ABL session can have a single session domain registry that you build at run time with information about each domain in which to authenticate and validate the user identity to set for the ABL session and (optionally) for its databases connections. For more information, see Setting up and using domain registries on page 123.

This ABL element...	Provides this identity management function...
CONNECT statement	This statement establishes a connection to one or more databases from within an ABL procedure or class. Given a user ID and password specified using the User ID (-U) and Password (-P) connection parameters for each database, this statement authenticates the user identity for each database connection; the user's domain must be authentication-enabled. If a specified user authentication is successful, the statement establishes the database connection with the specified identity; otherwise, the connection fails. If no -U and -P parameters are specified for a database, the statement establishes the database connection with the **OpenEdge default** blank ("") or operating system process user identity, and uses blank only if the database allows connection to the blank user identity; otherwise, the connection fails. For each established database connection, the statement internally generates a sealed client-principal object to represent the connection identity (either using an authenticated or OpenEdge default user identity), which you can return using the GET-DB-CLIENT function.[4] For more information, see Establishing database connection identity on page 135.
CREATE CLIENT-PRINCIPAL statement	Creates a client-principal object that you can use as a security token to start and maintain a given user login session. For more information, see Client-principal objects on page 111.
GET-DB-CLIENT function	This function returns a copy of the sealed client-principal object used to set the current user identity for a specified database connection. You can use this client-principal in an SSO operation to set another database connection or ABL session identity, unless the object represents the **OpenEdge default** blank ("") or operating system process user identity.[4] For more information, see Establishing database connection identity on page 135.

[4] If a sealed client-principal object represents the **OpenEdge default** blank or OS process user identity (set for a database connection established **without** using the -U connection parameter), you **cannot** use the object in an SSO operation to set the blank or OS process identity for any ABL session or database connection. However, if the object represents a blank or OS user identity that you have **authenticated** (using the -U connection parameter, a call to the SETUSERID function, or from calling the SET-CLIENT( ) method or SET-DB-CLIENT function on an unsealed client-principal), you **can** use the client-principal in SSO operations to set the blank or OS user identity, both for ABL sessions and database connections.

This ABL element...	Provides this identity management function...
`Progress.Lang.OERequestInfo` class	OpenEdge instantiates instances of this built-in class in order to pass client context information between an application server client and the application server sessions executing on behalf of the client running in any session model. OpenEdge automatically instantiates this class whenever a CREATE SERVER statement executes on an ABL client. Instances are accessible for both the client request and the application server response using attributes of the server object handle on the ABL client and attributes of the `SESSION` system handle on the application server session. One of the members of this class is a `ClientContextId` property that you can use to identify context in any ABL session executing for a given client request in a multi-tier application. The ABL session (application server client or sesion) can then use the property value to key a context database that maintains a consistent context shared by all sessions involved in the request.`Progress.Lang.OERequestInfo` also includes `GetClientPrincipal( )`and `SetClientPrincipal( )`, which are methods for accessing the client principal. For more information, see Establishing and managing identity for multi-tier applications on page 145 and Creating and managing unique object identities on page 154.
`SECURITY-POLICY` system handle	This object provides methods for building a single session domain registry for authenticating a user identity for both the ABL session and all existing OpenEdge database connections.[2], [3] The object also provides methods for setting and returning the session identity. For more information, see Setting up and using domain registries on page 123.

This ABL element...	Provides this identity management function...
`SET-CLIENT( )` method of the `SECURITY-POLICY` system handle	Given an unsealed client-principal object, this method authenticates the initialized user identity against the session domain registry; the user's domain must be authentication-enabled. If successful, it seals the client-principal and sets the ABL session identity to the authenticated identity; it then uses SSO to set the same user identity for any existing database connection that does not already have a connection identity assigned using the `SET-DB-CLIENT` or `SETUSERID` function. Given a sealed client-principal object, this method uses SSO to validate the associated user identity against the session domain registry. If successful, it sets the ABL session identity to the validated identity; it then uses SSO to set the same user identity for any existing database connection that does not already have a connection identity assigned using the `SET-DB-CLIENT` or `SETUSERID` function.[3] For more information, see Establishing ABL session identity on page 141.
`SET-DB-CLIENT` function	Given an unsealed client-principal object and a connected OpenEdge RDBMS, this function authenticates the user identity initialized for the object; the user's domain must be authentication-enabled. If successful, the function seals the client principal and sets the database connection to the authenticated identity. Optionally, the function can attempt to set the authenticated identity on all connected databases. Given a sealed client-principal object and a connected OpenEdge RDBMS, this function uses an SSO operation to validate the associated user identity against the database trusted domain registry and set the validated identity as the new database connection identity.[2]Optionally, the function can be called to validate and set the new identity on all connected databases. For more information, see Establishing database connection identity on page 135.

This ABL element...	Provides this identity management function...
SETUSERID function	Given a specified user ID and password, this function authenticates the specified user identity for an existing single database connection; the user's domain must be authentication-enabled. If successful, the function sets the authenticated identity as the current identity for the database connection, and internally generates a sealed client-principal object to represent that identity, which you can return using the GET-DB-CLIENT function. For more information, see Establishing database connection identity on page 135.
USERID function	Returns the connection identity of the specified database. This is also the default auditing identity for the database unless an option is set to use the ABL session identity. If the database is multi-tenant, the value returned is a fully-qualified user ID; otherwise, it is the account user name only. For more information, see Establishing database connection identity on page 135.

The actual ABL required for managing identities depends on your application security architecture, how you authenticate a given user ID, and the type of identity it is intended to assume. For more information on domain registries and how they are used in OpenEdge, see the sections on run-time domain configuration in *Introduction to Identity Management*.

Client-principal objects

A client-principal is a handle-based object that functions as the security token in an ABL application. As such, each instance of a client-principal object encapsulates a single user identity for an OpenEdge database connection, an ABL session, or an entire application. You can explicitly create a client-principal object that you initialize to authenticate a user identity, or you can obtain a copy of an existing client-principal object that represents the user identity already established for a given database connection or ABL session.

If you do not explicitly create a client-principal object for a database connection identity, OpenEdge implicitly creates one to represent the authenticated or default user identity for the connection. The ABL session identity is undefined unless you set the session identity using a client-principal object that you create or that you obtain from a database connection or other ABL session with an established identity.

You can thus use an instance of the object to assert user credentials to be authenticated as a user identity and to establish a user login session for the authenticated user identity. A user login session is a run-time construct that can be in any one of several states, indicated by the LOGIN-STATE and STATE-DETAIL attributes on the client-principal object handle (see Table 12: Client-principal object attributes are read-only on page 116). These states are controlled by methods and other attribute settings on the client-principal object handle, in addition to other ABL elements that operate on the object, such as the SET-DB-CLIENT function and SET-CLIENT( ) method on the SECURITY-POLICY system handle. Such states include whether the session is logged in or out, or has been suspended, expired, or otherwise failed. Thus, a client-principal object, provides the following application features:

- Maintains a run-time user login session that can be optionally recorded in an OpenEdge RDBMS and also as part of an audit trail if auditing is enabled and auditing policies for the database permit

- Contains a variety of information about the user login session, such as the user ID and unique session ID, a description of the authentication system (domain), a description of the login environment, the state of the login session, a unique identifier for the session, and other information, some of which are optional, depending on how you use the client-principal object
- Can be used to authenticate or validate a user identity against a trusted domain registry, and there by assign the identity to a particular OpenEdge context according to the type of domain registry (local database or session registry) and associated database option settings
- Can transport an asserted or authenticated user identity between different ABL session contexts in the same application environment

With all of its features, the ultimate purpose of a client-principal object is to enable your application to authenticate a user identity once and set that same identity many times, in many different places, thus achieving the goal of single sign-on for all database connections and sessions in your ABL application.

Unsealed and sealed client-principals

As noted previously (see Elements of an application authentication model on page 98), a security token can be unsealed or sealed. An unsealed client-principal is the initial state of any client principal object that you create using the `CREATE CLIENT-PRINCIPAL` statement (see Table 10: ABL elements for managing user identities on page 106). You can initialize and change the user identity (including all writeable attributes and data) of an unsealed client-principal. Sealing the client-principal puts the object in a state that prevents the identity it represents (or any of its attributes and data) from being changed, which allows the object to represent that identity with a high degree of reliability.

An unsealed client-principal object can be sealed:

- As part of any OpenEdge-performed user authentication operation using the `SET-DB-CLIENT` function or the `SET-CLIENT` method on the `SECURITY-POLICY` system handle (see Table 10: ABL elements for managing user identities on page 106)
- As an explicit operation that your ABL application can perform by executing the `SEAL( )` method on the object (see Table 13: Client-principal object methods on page 118)
- By the AVM after a callback has successfully completed authentication

A sealed client-principal can be in one of several different states, which corresponds to the state of the user identity it represents and is indicated by the `LOGIN-STATE` attribute (see Table 12: Client-principal object attributes are read-only on page 116). This state determines how the object can be used to represent its identity in your application and can also be reflected in any audit trail for the identity.

When a sealed client principal is in the LOGIN state, the object can be used in an SSO operation, which attempts to assign the user identity it represents to a given ABL session or database connection. The SSO operation first validates that the object has been sealed using a domain access code found in the session's or connection's trusted domain registry. If the validation is successful, the session or connection is then assigned the object's user identity.

Client-principal object attributes

An unsealed client-principal object provides several attributes that you can, or must, set for various functional and informational purposes. Whether or not you set them, writeable client-principal attributes become read-only once the object is sealed.

Before sealing a client-principal object that you have created, you must set several OpenEdge-required attributes and several optional attributes (depending on your application), as described and listed in the following table. Unless otherwise noted, these attributes hold `CHARACTER` values. All references to a "user" indicate the user associated with the user identity that this client-principal object represents.

Table 11: Client-principal object attributes you can set

This attribute...	Sets this value...
Required attributes	
DOMAIN-NAME	(Required) The name of the OpenEdge security domain in which the user identity is to be authenticated. This domain identifies the authentication system used to authenticate the non-qualified user ID specified by the USER-ID attribute. You must set this value before sealing the object and starting a login session. To seal the object and use it to establish identity, this value must be identical to one of the following and made available through a trusted domain registry: • The **Name** setting for a domain entry created in an OpenEdge database using the **Domains** dialog box in the Data Administration tool. This domain must also be enabled. • The Domain Name setting for a domain entry created in an OpenEdge database using the Domains editor of the **Database Administration Console**. This domain must also be enabled. • A domain name that you specify for a domain entry that you create at run time in the ABL session domain registry using the REGISTER-DOMAIN( ) method of the SECURITY-POLICY system handle (see Setting up and using domain registries on page 123.) For basic information on configuring domains and domain registries in OpenEdge, see *Introduction to Identity Management*. For information on accessing domains to authenticate a database connection identity, see Establishing database connection identity on page 135. For information on accessing domains to authenticate an ABL session identity, see Establishing ABL session identity on page 141. **Note:** Unless you create a session registry using the REGISTER-DOMAIN( ) method, it is up to the security administrator to ensure that the domain name you specify actually corresponds to the authentication system that you use to authenticate the user ID and that the domain is also configured with an appropriate domain access code.

This attribute...	Sets this value...
SESSION-ID	(Required) A value to uniquely identify the user login session represented by the client-principal object. This value must be set before sealing the object and starting a login session. You can set this to any value, but you should set it to a unique value to identify the login session in session audit trails. To obtain a unique value, you can use the ABL GENERATE-UUID function (see Creating and managing unique object identities on page 154). Otherwise, OpenEdge provides other ways to obtain a unique value. If you call the INITIALIZE( ) method on the client-principal without explicitly setting this value, OpenEdge automatically sets it to a unique value. If you want to associate the login session with a local application server session, you can also use the value of the SERVER-CONNECTION-ID attribute in a session-managed application or the ClientContextID property on the Progress.Lang.OERequestInfo object for any application session model (both available using the SESSION handle). For more information on when and how you might do this, see Establishing and managing identity for multi-tier applications on page 145.
USER-ID	(Required) The non-qualified user ID (account user name) of the user represented by this client-principal object. This is the user ID that must be authenticated in the domain specified by the DOMAIN-NAME attribute, whether OpenEdge, your application, or an ABL callback performs the authentication. You must set this value before sealing the object and starting a login session. **Note:** The security administrator must ensure that the OpenEdge domain specified by the DOMAIN-NAME attribute is configured with the authentication system used to authenticate this user ID.
Optional attributes	
AUDIT-EVENT-CONTEXT	When auditing is enabled, an application-defined value that OpenEdge stores in the _Event-context field of each audit event record generated by a client-principal SEAL( ), LOGOUT( ), or AUTHENTICATION-FAILED( ) method, or by an OpenEdge-performed user authentication using the SET-DB-CLIENT function or the SET-CLIENT( ) method on the SECURITY-POLICY system handle. You can set this value or leave it empty. If you do not set this value, for any ABL operation on the client-principal that generates an audit event, OpenEdge sets the attribute to any audit context value that the security administrator has specified for the user's domain (DOMAIN-NAME) configuration.
CLIENT-TTY	The name of the terminal display for the user's login session. You can set this value or leave it empty. OpenEdge does not otherwise set this attribute.
CLIENT-WORKSTATION	The name of the host workstation on which the user is working. You can set this value or leave it empty. OpenEdge does not otherwise set this attribute.

This attribute...	Sets this value...
DOMAIN-DESCRIPTION	A description of the domain in which the user specified by USER-ID is authenticated. You can set this value or leave it empty. If the attribute is empty, OpenEdge assigns any domain description that the security administrator has specified for the authentication domain configuration when you invoke the SEAL( ) or AUTHENTICATION-FAILED( ) method, or when you have OpenEdge perform the user authentication by invoking the SET-DB-CLIENT function, or by invoking the SET-CLIENT( ) method on the SECURITY-POLICY system handle.
DOMAIN-TYPE	The name of the authentication system used to authenticate the user specified by USER-ID. You can set this value or leave it empty. If the attribute is empty, OpenEdge assigns the value from the authentication system name configured for the user's domain (DOMAIN-NAME) when you invoke the SEAL( ) or AUTHENTICATION-FAILED( ) method, or when you have OpenEdge perform the user authentication by invoking the SET-DB-CLIENT function, or by invoking the SET-CLIENT( ) method on the SECURITY-POLICY system handle.
LOGIN-EXPIRATION -TIMESTAMP	A DATETIME-TZ value that specifies when the client-principal object (and any login session started with it) expires. At this point, OpenEdge sets the LOGIN-STATE attribute (see the following table) to "EXPIRED", the client-principal can no longer represent a user identity, and any started user login session for the object ends. OpenEdge detects that the specified time has passed only during ABL operations when it tries to seal the client-principal, validate the identity the object represents, or import the object from outside the ABL session. If you do not set this value, OpenEdge leaves it set to the Unknown value (?) and never sets the object to the EXPIRED login state.
LOGIN-HOST	The name of the host workstation on which the user's identity was authenticated. You can set this value or leave it empty. OpenEdge does not otherwise set this attribute.
PRIMARY-PASSPHRASE	The secret passphrase (or password) required to authenticate the user's ID (account name) specified by the USER-ID attribute. You **must** set this write-only value, along with all required attributes, **before** having OpenEdge authenticate the user identity (by passing the unsealed client-principal to the SET-DB-CLIENT function or to the SET-CLIENT( ) method on the SECURITY-POLICY system handle). For all other operations you invoke with the unsealed client-principal, including explicitly sealing the object with the SEAL( ) or AUTHENTICATION-FAILED( ) method, setting this value is entirely optional. Note that for a user authentication performed by your ABL application, this operation can be completely independent of the client-principal, which you typically seal to start a login session only **after** the authentication is successful. **Note:** Your application can only write values to this attribute, which can never be read. Also, any value you set is discarded without a trace by OpenEdge at the point the object is sealed, whether by your application following an application-performed authentication, or by OpenEdge following an OpenEdge-performed authentication, and whether or not the authentication is successful.

This attribute...	Sets this value...
QUALIFIED-USER-ID	The user's fully-qualified user ID, which consists of the USER-ID attribute (account user name) value followed by the DOMAIN-NAME attribute value separated by the '@' character (*account-name*@*domain-name*). If you set the QUALIFIED-USER-ID attribute, OpenEdge sets USER-ID and DOMAIN-NAME accordingly, and if you set USER-ID and DOMAIN-NAME, OpenEdge sets QUALIFIED-USER-ID accordingly. OpenEdge supports certain patterns for setting and reading this value, which set or return default values for the USER-ID and DOMAIN-NAME attributes, appropriately. For more information, see the entry for the QUALIFIED-USER-ID attribute in *ABL Reference*.
ROLES	A comma-separated string of domain roles for the user identity associated with this object. You can set this value or leave it empty. OpenEdge does not otherwise set this attribute.

Some client-principal attributes are read-only and can never be set, even for an unsealed client-principal. For a given client-principal object, OpenEdge sets the read-only values of these attributes on behalf of the application, depending on client-principal method calls and other operations.

The following table describes these read-only attributes (other than HANDLE, INSTANTIATING-PROCEDURE, and TYPE, which are common to all handle-based objects). For more information on how these attributes are set, see Table 10: ABL elements for managing user identities on page 106, the above table and the Table 13: Client-principal object methods on page 118 table.

Table 12: Client-principal object attributes are read-only

This attribute...	Returns this value...
DB-LIST	A comma-separated list of logical names of all OpenEdge multi-tenant databases for which a connection identity has been set using this sealed client-principal object in one or more ABL sessions. This list ignores all connections to non-multi-tenant databases.

This attribute...	Returns this value...
LOGIN-STATE	A CHARACTER value that represents the state of the client-principal object and any user login session started with it. The following describes the valid values for LOGIN-STATE and the possible transitions from one state to another: • "INITIAL" — (Default value) The state of an unsealed client-principal after it is created (either explicitly when using the CREATE CLIENT-PRINCIPAL statement or automatically when a CREATE SERVER statement executes on an ABL client), or after calling the INITIALIZE( ) method on an existing sealed or unsealed client-principal. From "INITIAL", LOGIN-STATE can transition to "LOGIN", "FAILED", "EXPIRED", or "LOGOUT". • "LOGIN" — The state when an unsealed client-principal is successfully sealed using the SEAL( ) method (see Table 13: Client-principal object methods on page 118), the SET-CLIENT( ) method on the SECURITY-POLICY system handle, or the SET-DB-CLIENT function (see Table 10: ABL elements for managing user identities on page 106). At this point OpenEdge starts a login session and the object can be used to represent an authenticated user identity.[5] From "LOGIN", LOGIN-STATE can transition to "EXPIRED", or "LOGOUT". • "EXPIRED" — The state when OpenEdge detects that the current date and time is later than a DATETIME-TZ setting of the LOGIN-EXPIRATION-TIMESTAMP attribute (see the above table). In that event, OpenEdge sets this value and seals the client-principal, which can no longer be used to represent a user identity. OpenEdge cannot transition to another state. • "FAILED" — The state after the AUTHENTICATION-FAILED( ) method is invoked on the unsealed client-principal, or after an OpenEdge-performed user authentication operation fails on the unsealed object using the SET-CLIENT( ) method or the SET-DB-CLIENT function. In that event, OpenEdge sets this value and seals the client-principal, which can no longer be used to represent a user identity. OpenEdge cannot transition to another state. • "LOGOUT" — The state after OpenEdge terminates the login session, and the sealed client-principal can no longer be used to represent a user identity. OpenEdge cannot transition to another state. • "SSO" — The state when the client principal was sealed by a non-OpenEdge user authentication process. From "SSO", LOGIN-STATE can transition to "EXPIRED", or "LOGOUT".

[5] OpenEdge also implicitly creates a client-principal object and sets this value when it successfully authenticates the user identity specified using the User ID (-U) and Password (-P) connection parameters (on the command line or in the CONNECT statement) and seals the object.

This attribute...	Returns this value...
STATE-DETAIL	A CHARACTER value that provides more detailed information about the most recent setting of the LOGIN-STATE attribute. OpenEdge also sets this attribute to any *reason* parameter value that you pass to the AUTHENTICATION-FAILED( ) method (see Table 13: Client-principal object methods on page 118).
SEAL-TIMESTAMP	A DATETIME-TZ value that represents the date and time when the client-principal object was sealed and a user login session was started (changing the object login state from INITIAL to LOGIN). OpenEdge sets this value when you authenticate a user identity and implicitly or explicitly seal an unsealed client-principal by calling the SET-DB-CLIENT function, the SET-CLIENT( ) method on the SECURITY-POLICY system handle (see Table 10: ABL elements for managing user identities on page 106), or the SEAL( ) method (see Table 13: Client-principal object methods on page 118).[5]

Client-principal object methods

After you create a client principal object, in addition to setting writeable attributes, you can set and read any number of application-defined properties using methods on the object handle. After all required attributes and application-defined properties have been set, you can invoke additional ABL methods and functions to start and manage a user login session. The following table describes all the methods that you can invoke on the client-principal object, organized according to usage. For more information on authenticating and setting user identity with a client-principal, see Authenticating and setting user identity on page 128.

Table 13: Client-principal object methods

This method...	Provides this function...
Application property set and get methods	
GET-PROPERTY(*property-name*)	Given the name of an application-defined property (*property-name*, expressed as a case-sensitive, quoted string), returns the string value assigned to that property using the SET-PROPERTY( ) method.
LIST-PROPERTY-NAMES( )	Returns a comma-separated list of the names for all application-defined properties set using the SET-PROPERTY( ) method.
SET-PROPERTY(*property-name*, *property-value*)	Defines a property in the client-principal object with a name (*property-name*) specified by a case-sensitive, quoted string and a value specified by a character expression (*property-value*). You can call this method only once for each property you define in the object, preventing the definition of multiple properties with the same name.

This method...	Provides this function...
Login session management methods	
AUTHENTICATION-FAILED ([*reason*])	Indicates that the user ID associated with an unsealed client-principal object (in the INITIAL state) cannot be authenticated. After calling this method, the LOGIN-STATE attribute is set to "FAILED", the client-principal object is invalidated, its attributes and properties cannot be changed, and the object is sealed but cannot be used to represent a user identity. Calling this method also writes the value of any character expression you provide for the *reason* parameter to the STATE-DETAIL attribute. If auditing is enabled and auditing policies permit, the method generates an audit event record to indicate a user login failure.
EXPORT-PRINCIPAL( )	Converts and returns a sealed or unsealed client-principal object, including all of its attribute and property settings, as a RAW value. You can then store this value or pass it as a remote procedure parameter that an application server session can use to create a copy of the object using the IMPORT-PRINCIPAL method. Thus, this method allows you to transport the asserted or authenticated user identity represented by a client-principal object from one ABL session to another.
IMPORT-PRINCIPAL(*expression*)	Given a RAW *expression* that represents a sealed or unsealed client-principal object (as output from the EXPORT-PRINCIPAL( ) method), sets the current handle to a copy of the exported client-principal object. You can then use the copy as appropriate for the exported object.
INITIALIZE(*qualified-user-id* [, *session-id* [, *expiration* [, *primary-passphrase*]]])	

This method...	Provides this function...
	Sets an existing unsealed or sealed client-principal object to the INITIAL state, with attribute values initialized from the following parameter list: • ***qualified-user-id*** — (Required) Sets the QUALIFIED-USER-ID attribute, which in turn sets the USER-ID and DOMAIN-NAME attributes. • ***session-id*** — If specified, sets the SESSION-ID attribute. By default, OpenEdge initializes SESSION-ID to a globally unique value (GUID). • ***expiration*** — If specified, sets the LOGIN-EXPIRATION-TIMESTAMP attribute to a date and time after which any use sets the object to the EXPIRED state. Otherwise, OpenEdge sets the Unknown value (?) and the object has no expiration date and time. • ***primary-passphrase*** — If specified, sets the PRIMARY-PASSPHRASE attribute for user authentication. Otherwise, OpenEdge sets the Unknown value (?) and no password is set for user authentication. All other attributes of the object are reset to their default values. Any prior list of application properties and their settings are removed from the object. This method allows you to re-use an existing client-principal for a different identity rather than creating an additional object to manage in the session. You can thus represent different consecutive identities with a single object that you need only delete once when all identity management with the object is complete.
LOGOUT( )	Indicates that the user ID associated with a sealed client-principal object (in the LOGIN state) has logged out of its user login session. After calling this method, the LOGIN-STATE attribute is set to "LOGOUT", the client-principal object is invalidated, its attributes and properties cannot be changed, and the object can no longer be used to represent a user identity. If auditing is enabled and auditing policies permit, the method generates an audit event record to indicate the user logout.
SEAL(*domain-access-code*)	

This method...	Provides this function...
	Given a character expression (*domain-access-code*), seals the unsealed client-principal object and starts a user login session for the user identity it represents. This method also generates an optional user login session record in all connected databases. Before calling this method, all the required attributes must be set (see Table 11: Client-principal object attributes you can set on page 113). Typically, use of this method assumes that your application has authenticated the user identity that it represents, though such authentication is not required. For users in an authentication-enabled domain, you can have OpenEdge authenticate and seal the client-principal in a single operation using the SET-CLIENT( ) method on the SECURITY-POLICY system handle or the SET-DB-CLIENT function. These methods also set the authenticated user identity for supported security systems (the ABL session or its current database connections). Sealing a client-principal generates a digital seal that is stored with the object in the form of a message authentication code (MAC), which is based on the value of *domain-access-code* (MAC key) and the contents of the object. This seal is used to validate the client-principal object before setting user identities with it. To generate a validating seal, you must ensure that the value of *domain-access-code* is identical to the access code defined for the security domain that is specified by the DOMAIN-NAME attribute and stored in the trusted domain registry (see Authenticating and setting user identity on page 128). Calling this method also sets the SEAL-TIMESTAMP attribute and checks the value of the LOGIN-EXPIRATION-TIMESTAMP attribute (see Table 11: Client-principal object attributes you can set on page 113). If the client-principal has expired, OpenEdge sets the LOGIN-STATE attribute to "EXPIRED", making the object unusable to represent a user identity. Otherwise, it sets the attribute to "LOGIN", and if auditing is enabled and auditing policies permit, it generates an audit event record to indicate a valid user login. After calling this method successfully, you can pass the sealed client-principal object in a call to the SET-DB-CLIENT function or the SET-CLIENT( ) method on the SECURITY-POLICY system handle, which then validates the user identity the object represents and assigns the validated identity to all supported security systems.
TENANT-ID(*db-exp*)	

This method...	Provides this function...
	Returns the tenant ID that is associated with the user identity sealed in the client-principal and assigned to a connection to the multi-tenant database specified by *db-exp*. The current client-principal object contains the tenant ID associated with the specified database. In order to contain this tenant ID, the current client-principal must have been used to assign its user identity to the database connection in the current ABL session, or in another ABL session from which the object was returned to the current ABL session. Once the connection identity is assigned, OpenEdge stores the ID for the tenant associated with the connection identity in the client-principal. However, the specified database does not have to be currently connected to return the associated tenant ID. Once a tenant ID for a database connection is stored in the client-principal, it cannot be removed and can be returned for the duration of the corresponding user login session.
`TENANT-NAME( db-exp )`	Returns the tenant name that is associated with the user identity sealed in the client-principal and assigned to a connection to the multi-tenant database specified by *db-exp*. The current client-principal object contains the tenant name associated with the specified database. In order to contain this tenant name, the current client-principal must have been used to assign its user identity to the database connection in the current ABL session, or in another ABL session from which the object was returned to the current ABL session. Once the connection identity is assigned, OpenEdge stores the name for the tenant associated with the connection identity in the client-principal. However, the specified database does not have to be currently connected to return the associated tenant name. Once a tenant name for a database connection is stored in the client-principal, it cannot be removed and can be returned for the duration of the corresponding user login session.
`VALIDATE-SEAL( [ domain-access-code ] )`	

This method...	Provides this function...
	Validates the MAC seal generated for a client-principal object by the SEAL( ) method. To call this method on a sealed client-principal object, the LOGIN-STATE attribute must be set to "LOGIN". You typically validate the seal on a sealed client-principal object after you have imported it (using the IMPORT-PRINCIPAL( ) method) from some external storage that is beyond the exclusive control of your application or whose integrity is otherwise questionable. To successfully validate the seal, you must provide the method with a domain access code value in one of the following ways: • Pass it a parameter (*domain-access-code*) whose value is identical to the domain access code used to seal the object. • Do not pass a value and ensure that the session domain registry contains a domain entry matching the security domain defined for the client-principal object by the DOMAIN-NAME attribute and that the entry includes a domain access code whose value is identical to the access code used to seal the object (see Setting up and using domain registries on page 123). Calling this method also checks the value of the LOGIN-EXPIRATION-TIMESTAMP attribute (see Table 11: Client-principal object attributes you can set on page 113). If the client-principal object has expired, ABL sets the LOGIN-STATE attribute to "EXPIRED", making the object unusable to represent a user identity.

Setting up and using domain registries

When OpenEdge authenticates a user identity or validates an authenticated identity through SSO, it first retrieves the OpenEdge domain configuration identified by user credentials or a client-principal to obtain the following information:

- **User authentication** — The authentication system to authenticate a user's credentials and (if successful) the domain access code to seal the client-principal object representing the authenticated identity to be assigned to database connection or ABL session.
- **SSO** — The domain access code to validate that it is identical to the access code used to seal a given client-principal in order to assign the identity the object represents to a database connection or ABL session.

To retrieve the domain in either case, OpenEdge searches a trusted domain registry for a domain entry that matches the user's domain.

Identifying the trusted domain registry

OpenEdge provides two registry options to serve as the trusted domain registry in an ABL application:

- **Local database domain registry** — The default domain registry used to authenticate or validate the user identity for a database connection. OpenEdge automatically creates a local database domain registry from the domains that are configured and enabled for use in the OpenEdge database when the database server (or client for a self-service database) starts up. Operations to manage the identity for all connections to this

database use this local domain registry unless a database option is set to use the ABL session (application) domain registry instead.

- **ABL session domain registry** — The domain registry used to authenticate or validate the user identity for an ABL session. This registry can also be used to authenticate or validate the identity for an existing database connection when a database option is set to use the application domain registry instead of the local database registry. For an ABL session, there is no default session domain registry. Your application must always build the session registry at run time using appropriate ABL mechanisms before it performs any identity management operations on the ABL session.

For basic information on configuring domains and domain registries in OpenEdge, see *Introduction to Identity Management*. The following section describes the ABL features for building and using an ABL session domain registry.

Building and using a session domain registry

Typically, an ABL application builds any required session domain registry for an ABL session (including any application server sessions) when the ABL session first starts up. For an ABL client, this is typically a function of a startup procedure specified using the Startup Procedure (`-p`) startup parameter. For an application server session, this is typically a function of the Startup configuration procedure for application servers.

Note: If you do not build a session domain registry, you cannot establish an ABL session or application user identity. However for a default database configuration, you can always establish a database connection identity, with or without a session registry, in order to authorize access to database tables and fields. For more information on identity usage, see User identity applications on page 105.

OpenEdge supports two approaches for building a session domain registry, using:

- The contents of an existing local database domain registry (recommended)
- Domain entries that you create directly within the ABL session

You can only use one of these approaches to build a domain registry for a given ABL session, and using the contents of an existing local database domain registry is the most secure approach, as described later in this section.

The `SECURITY-POLICY` system handle supports methods for each approach to building a session domain registry, as described in the following table.

Table 14: Methods for building a session domain registry

This SECURITY-POLICY method...	Provides this function...
`LOAD-DOMAINS( integer-expression | logical-name | alias )`	Builds the session domain registry from the local database domain registry of a specified OpenEdge database connection. (The *integer-expression* value specifies the database by its order of connection in the ABL session.) You can call this method multiple times to refresh the session registry from the latest contents of the **same** local database domain registry only. Calling this method using a different database, or using a different connection to the same database, raises an error. You can use the session domain registry built with this method to: • Authenticate and set the ABL session identity and the connection identity of a properly configured OpenEdge database using an unsealed client-principal (through OpenEdge-performed user authentication or ABL callback authentication) or using a sealed client-principal (through OpenEdge-performed SSO). • Seal an unsealed client-principal using the client-principal `SEAL( )` method. If you call either the `LOCK-REGISTRATION( )` method or the `REGISTER-DOMAIN( )` method successfully, any call to this method fails.[6] Once you call this method successfully, any call to `LOCK-REGISTRATION( )` or `REGISTER-DOMAIN( )` fails. **Note:** This method is more secure for building the session domain registry than using the `REGISTER-DOMAIN( )` method, and it is the only way to build a session registry that can be used to authenticate or validate the connection identity for an OpenEdge multi-tenant database.
`LOCK-REGISTRATION( )`	

This SECURITY-POLICY method...	Provides this function...
	Locks the session domain registry against any further update using the `REGISTER-DOMAIN( )` method.[6] If you call this method before calling `REGISTER-DOMAIN( )`, the locked session registry is empty, but usable. You **must** call this method before you can validate user identities using a session registry built using the `REGISTER-DOMAIN( )` method. If you call the `LOAD-DOMAINS( )` method successfully before calling this method, any call to this method fails. Once you call this method successfully, the registry is usable and any call to `LOAD-DOMAINS( )`or `REGISTER-DOMAIN( )` fails.

`REGISTER-DOMAIN`
`( domain-name ,`
`access-code`
`[, domain-description`
`[, domain-type]] )`	

[6] Once the session domain registry is locked, it cannot be replaced or updated during the ABL session. The only way to change it is to restart the session and rebuild the registry.

This SECURITY-POLICY method...	Provides this function...
	Creates a single domain entry in the session domain registry from the parameters passed to the method: • ***domain-name*** — Specifies the name of an OpenEdge domain, which must match the DOMAIN-NAME property of any client-principal object that you seal using this domain entry (see Table 11: Client-principal object attributes you can set on page 113). • ***access-code*** — Specifies the value to use as a MAC key for sealing a client-principal object, or for validating a client-principal object that has been sealed, against this domain entry (see Table 13: Client-principal object methods on page 118). • ***domain-description*** — Allows an informal description of the domain. Otherwise, the value is blank (""). • ***domain-type*** — Allows you to specify the name of an authentication system for the domain. Otherwise, the value is blank (""). You can call this method multiple times to build a session domain registry. You can then use this registry to seal an unsealed client-principal using the client-principal SEAL( ) method and to set a session identity or a connection identity for an OpenEdge database (**non-multi-tenant only**) using a sealed client-principal. However, OpenEdge can **never** use a session registry built with this method to authenticate an identity asserted with an unsealed client-principal (see LOAD-DOMAINS( ) described in this table). If you call either the LOCK-REGISTRATION( ) method or the LOAD-DOMAINS( ) method successfully, any call to this method fails. Once you call this method successfully, any call to LOAD-DOMAINS( ) fails. **Note:** The *domain-description* and *domain-type* values do **not** have to match the DOMAIN-DESCRIPTION and DOMAIN-TYPE attributes in a client-principal object in order to validate its identity against a domain entry.

The REGISTER-DOMAIN( ) method provides the flexibility to dynamically define an authentication domain not otherwise available in a database domain registry. However, until you call the LOCK-REGISTRATION( ) method, you run the risk of a security breach by allowing the possible registration of rogue domains in the session registry. The LOAD-DOMAINS( ) method minimizes this risk by leveraging secure mechanisms both to manage the contents of an existing database domain registry (by a security administrator) and to load the latest contents of this registry (and **only** this registry) into a given session domain registry.

For more information on configuring domains in an OpenEdge database, see:

- *Introduction to Identity Management*
- The Data Administration online help
- *Database Tools*
- *Configure Database Multi-tenancy with OpenEdge Management*

For information on building a session domain registry, see Establishing ABL session identity on page 141 in this manual.

Authenticating and setting user identity

This section introduces and describes the key features of all the mechanisms that ABL supports to authenticate and set a user identity. For some examples of their use, see the following sections.

OpenEdge supports the following ABL elements that allow OpenEdge or an ABL application to authenticate and set a user identity:

- User ID (-U) and Password (-P) database connection parameters on page 128
- SET-DB-CLIENT function on page 129
- SETUSERID function on page 130
- SET-CLIENT( ) method on the SECURITY-POLICY system handle on page 130
- SEAL( ) method on the client-principal object handle on page 131

User ID (-U) and Password (-P) database connection parameters

This is the minimum command-line syntax to connect a single database, where *AVM-startup* is an ABL client startup command (or executable), such as `mpro` (or `prowin32`), *userid* and *password* specify a user ID and passphrase (user account password), and *physical-dbname* identifies a database to be connected:

Syntax

```
AVM-startup -db physical-dbname -U userid -P password
```

This is the equivalent `CONNECT` statement syntax:

Syntax

```
CONNECT { physical-dbname "-U userid -P password" } |
        {"-db physical-dbname -U userid -P password" }
```

When provided on the AVM startup command line or in options of the CONNECT statement (see Database connections on page 36), OpenEdge authenticates the specified user credentials to connect a particular OpenEdge database. If OpenEdge successfully authenticates the specified identity, along with creating the database connection with that identity, it creates and seals a client-principal object in the LOGIN state with the QUALIFIED-USER-ID and PRIMARY-PASSPHRASE attributes set from the input user credentials and all other properties and attributes of the object set to default values. Sealing the object with user credentials provided using the -U/-P connection parameters also generates an auditable event to start a user login session. Using either the command-line or the CONNECT statement, OpenEdge always authenticates the identity and seals the client-principal object using the local domain registry for the specified database.

Caution: If you connect an OpenEdge database without specifying -U/-P, OpenEdge automatically connects the database with a default OpenEdge identity and also creates and seals a client-principal with that default identity, gaining access to the database without authentication. However, you cannot use a client-principal sealed with a default identity in an SSO operation to assign the identity to a database connection or ABL session, because the domain access code used to seal the object is not available in any trusted domain registry used to validate the object identity. To ensure that a database connection started from the command line has an authenticated identity, you can require input of a user ID and password and authenticate them using one of the ABL elements that sets the identity for an established connection.

SET-DB-CLIENT function

This is the basic syntax, where *client-principal-handle* is the handle to an unsealed or sealed client-principal object and *database-connection* optionally identifies an established database connection:

Syntax

```
SET-DB-CLIENT( {client-principal-handle}[ , database-connection] )
```

When you provide user credentials as input to the function in an **unsealed** client-principal (in the INITIAL state), OpenEdge or an ABL callback authenticates the user identity for a specified existing OpenEdge database connection or for the first database connection established in the current session (if no database connection is specified). If OpenEdge or an ABL callback successfully authenticates the identity, the AVM seals the client-principal object in the LOGIN state and sets the connection to the specified identity. Sealing the object with this function also generates an auditable event to start a user login session. If no database connection is specified, it then uses the sealed client-principal in an SSO operation on each **additional** connected OpenEdge database to validate and (if valid) set the connection identity.

When you provide a **sealed** client principal (in the LOGIN state) as input to the function, OpenEdge then uses it in an SSO operation on the specified database connection, or if not specified, on every database connection in the current session to validate and (if valid) set the connection identity of the given database.

When the SET-DB-CLIENT function executes for each database connection, OpenEdge authenticates and seals the identity specified in the unsealed client-principal, or performs an SSO operation on the sealed client-principal, using either the local database domain registry for the given connection (by default) or the current session domain registry when a given database has its option set to use the application domain registry.

Note: Setting the identity for a database connection using this function locks out the SECURITY-POLICY:SET-CLIENT( ) method from setting the connection identity. You can unlock the connection for SET-CLIENT( ) access by calling SET-DB-CLIENT with *client-principal-handle* passed as the Unknown value (?).

SETUSERID function

This is the basic syntax, where *userid* and *password* pass in a user ID and passphrase (user account password) and *logical-dbname* optionally identifies an established database connection:

Syntax

```
SETUSERID( userid , password [ , logical-dbname ] )
```

OpenEdge authenticates user credentials to set the identity for a particular OpenEdge database connection, all based on parameters passed as input to the function. If you do not identify a database connection, OpenEdge attempts to locate and use a connection to the database for which the current procedure or class was compiled.

For backward compatibility, this function **always** authenticates the specified user credentials against the internal OpenEdge user accounts defined in the database `_User` table. If OpenEdge successfully authenticates the specified identity, it assigns the identity to the database connection, and creates and seals a client-principal object in the LOGIN state with the `QUALIFIED-USER-ID` and `PRIMARY-PASSPHRASE` attributes set from the input user credentials and with all other properties and attributes of the object set to default values. Sealing the object with this function does **not** generate an auditable event to start a user login session, but does generate an auditable event for the change in user identity.

When the `SETUSERID` function executes, OpenEdge authenticates and seals the identity in the client-principal object using either the local database domain registry for the connection (by default) or the current session domain registry when the connected database has its option set to use the application domain registry.

Note: Setting the identity for a database connection using this function locks out the `SECURITY-POLICY:SET-CLIENT( )` method from setting the connection identity. You can unlock the connection for `SET-CLIENT( )` access by calling the `SET-DB-CLIENT` function with its *client-principal-handle* parameter passed as the `Unknown value (?)`.

SET-CLIENT( ) method on the SECURITY-POLICY system handle

This is the basic syntax, where *client-principal-handle* is the handle to an unsealed or sealed client-principal object:

Syntax

```
SECURITY-POLICY:SET-CLIENT( client-principal-handle )
```

When you provide user credentials as input to the function in an **unsealed** client-principal (in the INITIAL state), OpenEdge authenticates the user identity for the current ABL session. If OpenEdge successfully authenticates the identity, it seals the client-principal object in the LOGIN state and sets the session to the specified identity. Sealing the object with this method also generates an auditable event to start a user login session. It then uses the object in an SSO operation to validate and (if valid) set the same identity on each established OpenEdge database connection whose identity has not **already** been set using the `SET-DB-CLIENT` or `SETUSERID` function (which *locks out* `SET-CLIENT( )` from setting the identity for this connection).

When you provide a **sealed** client principal (in the LOGIN state) as input to the function, OpenEdge then uses it in an SSO operation on the current ABL session to validate and (if valid) set the session identity. If the SSO operation on the ABL session is valid, OpenEdge then uses the object in an SSO operation to validate and (if valid) set the same identity on each established OpenEdge database connection that is not already locked out by using the SET-DB-CLIENT or SETUSERID function on the connection.

Note: You can unlock SET-CLIENT( ) access to a specified database connection, or all database connections, by executing SET-DB-CLIENT with its *client-principal-handle* parameter set to the Unknown value (?).

When the SET-CLIENT( ) method executes, OpenEdge authenticates and seals the identity specified in an unsealed client-principal, or performs an SSO operation on a sealed client-principal, first using the session domain registry to set the ABL session identity. Then, for each available database connection, OpenEdge performs the SSO operation using either the local database domain registry for the connection (by default) or the current session domain registry when a given database has its option set to use the application domain registry.

SEAL( ) method on the client-principal object handle

This is the basic syntax, where *client-principal-handle* is the handle to an unsealed client-principal object and *domain-access-code* is typically a MAC key value (see Table 13: Client-principal object methods on page 118):

Syntax

```
client-principal-handle:SEAL( domain-access-code )
```

This method operates only on an **unsealed** client-principal (in the INITIAL state) and attempts to seal the object in the LOGIN state if it finds a domain entry in the session domain registry whose domain name matches the client-principal DOMAIN-NAME attribute value, and whose domain access code matches the *domain-access-code* value passed to the method. This operation assumes that your application has **already** authenticated the user identity initialized in the client-principal object. However, OpenEdge has no way to verify this; it only verifies that the object is initialized properly and that there is a matching domain name and access code in the session registry. If the values match, it then seals the client-principal using the domain access code. Sealing the object with this method also generates an auditable event to start a user login session.

Caution: Application-performed user authentication is inherently much less secure than OpenEdge-performed user authentication. Although your application might have access to a secure source of user accounts, it is entirely responsible for secure access to those accounts and for authenticating a user identity against them. Most importantly, by sealing a client-principal with a given user identity, the application is certifying that the identity it represents is authenticated and valid, regardless if and how the application has authenticated the identity. This certification depends on direct application access to a valid domain access code, which has the potential to be hacked at least until the application seals the client-principal and destroys the value of the access code that it has used to do so. On the other hand, during OpenEdge-performed user authentication, all access to user credentials, domain access codes, and valid user accounts is managed by OpenEdge in a secure processing environment that relies on industry best practices for identity management.

Note that this function does **not** assign the identity sealed in the client-principal to any ABL resource. You must use the successfully sealed client-principal in a separate SSO operation that you execute using either the SET-CLIENT( ) method or the SET-DB-CLIENT function in order to validate and (if valid) assign the identity to the current ABL session and any indicated database connections established in the session.

Initializing a client-principal for user authentication and SSO

Some user authentication mechanisms require you to explicitly create and initialize a client-principal object at appropriate points in the authentication process. Typically, you create a client-principal for initialization using the `CREATE CLIENT-PRINCIPAL` statement (see Table 10: ABL elements for managing user identities on page 106).

How you initialize a client-principal object for user authentication depends on whether OpenEdge (or an ABL callback) or your application is performing the user authentication. If OpenEdge performs the user authentication (when you execute either the `SET-DB-CLIENT` function or the `SET-CLIENT( )` method on the `SECURITY-POLICY` system handle), you must first initialize the client-principal with user credentials and additional information that OpenEdge requires by setting appropriate values for the following client-principal attributes (see Table 11: Client-principal object attributes you can set on page 113):

- `USER-ID`
- `DOMAIN-NAME`
- `PRIMARY-PASSPHRASE`
- `SESSION-ID`

If your application performs the user authentication, the required user credentials depend entirely on the user account system that your application uses to authenticate its user identity. If your application user account system requires different user credentials than OpenEdge requires, you might have to derive the settings of the `USER-ID` and `DOMAIN-NAME` attributes from these credentials. Also for application-performed user authentication, you typically initialize and seal the client-principal only **after** the user authentication has completed in order to reflect its success or failure. So, you need only set the following attributes to initialize the client-principal:

- `USER-ID`
- `DOMAIN-NAME`
- `SESSION-ID`

You have no need to set the `PRIMARY-PASSPHRASE` attribute, because it is no longer needed following user authentication, and in any case, OpenEdge discards its value without a trace when you seal the object using either the `SEAL( )` method (for a successful user authentication) or the `AUTHENTICATION-FAILED( )` method (for an unsuccessful user authentication).

To optimize client-principal initialization, OpenEdge provides the `INITIALIZE( )` method (see Table 13: Client-principal object methods on page 118), which allows you to set all the required attributes in a single operation. Using this method, you can also set the `LOGIN-EXPIRATION-TIMESTAMP` attribute, which sets an expiration date and time for the client-principal. Another optimization with this method is that you can re-initialize an existing client-principal that is sealed in **any** state back to the INITIAL state. In a single operation, this implicitly deletes the existing object and creates a new object initialized with the specified identity settings that you can then authenticate and seal as appropriate.

Also prior to any operation that seals the client-principal, in addition to the required attribute settings, you can individually set any optional attribute values (see Table 11: Client-principal object attributes you can set on page 113) and also set any application-defined properties that you want to associate with the login session using the `SET-PROPERTY( )` method on the client-principal object handle (see Table 13: Client-principal object methods on page 118).

Managing the client-principal after establishing identity

After the identity for a new or existing database connection is established, you can return a copy of the client-principal object associated with the connection identity by invoking the `GET-DB-CLIENT` function on the connection. This is especially useful to return a client-principal implicitly created by OpenEdge for the `-U/-P` connection parameters or the `SETUSERID` function so you can use it in an SSO operation to set the identity of another database connection or the current ABL session. You can similarly return a copy of the client-principal object used to set the current ABL session identity using the `GET-CLIENT( )` method on the `SECURITY-POLICY` system handle.

Note that a given ABL session can create and maintain multiple client-principal objects for multiple user identities. However, for a given application, whether it is self-service, client-server, or multi-tier, a given application session typically maintains only **one client-principal object per user**.

Note: Progress Software recommends that any ABL session maintain **only one client-principal object per user identity** for all ABL sessions and database connections of an application. A context server might maintain a context store that includes a copy of this client-principal for each user of a multi-tier application, in addition to the user's application state and data. However, when creating or returning application context, this context server typically does so for only one user per request.

Note that all operations that seal a client principal, except the `SETUSERID` function, also start an associated user login session whose state is indicated by the value of the `LOGIN-STATE` attribute (see Table 12: Client-principal object attributes are read-only on page 116). This login session is most useful for multi-tier applications that use the client-principal to pass an audit identity among several ABL sessions. In any case, the transition to different login states generates auditable events that can be recorded in each database connected in a session with the same audit identity.

Configuring login sessions

If you want the user login session recorded in an OpenEdge RDBMS, you must select the **Record Authenticated Client Sessions** option in the **Database Options** dialog box of the Data Administration tool or the character-mode Data Dictionary. With client session recording enabled, for each user login session that you successfully start, OpenEdge generates a single `_client-session` record keyed on the user login session ID.

Note: The user login session ID, which uniquely identifies a user login session, is different from any user identity established for the session using a client-principal. Multiple unique user login session records can exist for the same user identity, whether the it is a session identity or a database connection identity.

The contents of this record come from selected attributes of the client-principal object handle, selected ABL session attributes, and, if auditing is enabled, values derived from active audit policy data. For more information on the `_client-session` table and relevant audit policy data, see *Introduction to Security and Auditing*.

Note: If the database is also audit-enabled, the recorded user login session ID identifies an audit event context used to record several user login session audit events. For more information on using audit event context, see Auditing on page 241.

Starting and maintaining user login sessions

A client (user) login session begins at the point that a client-principal object that represents a user identity is sealed in the LOGIN state after a successful user authentication (except using the SETUSERID function; see SETUSERID function on page 130). If auditing is enabled, and the authenticated identity is an auditing identity, this also generates a **login audit event** that is recorded in each connected OpenEdge database according to its audit policy settings. The login audit event thus records the start of the login session, which continues with all activity associated with the same login session identity up to the point of any event that terminates the login session in any ABL session (for multi-tier) of the application. The terminating event for a login session can include:

- **A logout audit event** — Generated when the LOGOUT( ) method is invoked on a client-principal that represents the login session
- **An expired audit event** — Generated at the point a client-principal that represents the login session is used when the current date and time exceeds the expiration time stamp set for the client-principal using the LOGIN-EXPIRATION-TIMESTAMP attribute on the client-principal.

You can start and manage user login sessions in a similar manner for any type of identity that you create. The following procedure applies generally in all cases.

To start and maintain a user login session:

1. Successfully authenticate a user identity using any ABL mechanism other than the SETUSERID function. For more information, see Authenticating and setting user identity on page 128. For examples, see:

   - Establishing database connection identity on page 135
   - Establishing ABL session identity on page 141

   Whether OpenEdge seals the client-principal or your application seals the client-principal, at that point, it also sets the read-only SEAL-TIMESTAMP attribute (see Table 12: Client-principal object attributes are read-only on page 116), which records the effective user authentication time for the user login session.

2. Given that the login is successful, whether created by OpenEdge or your applications, you can then use the sealed client-principal object in an SSO operation to assert and validate the user identity for the user login session against any OpenEdge database connection or ABL session.
3. If you need to transport the user login session between the ABL sessions of a multi-tier application, you can export the client-principal object in one ABL session context using the EXPORT-PRINCIPAL( ) method and import it into another session context using the IMPORT-PRINCIPAL( ) method on a different client-principal object, validating the result, if necessary, using the VALIDATE-SEAL( ) method on the newly imported client-principal object (see Table 13: Client-principal object methods on page 118). For more information on when you might need to do this, see Establishing and managing identity for multi-tier applications on page 145.
4. To invalidate a user login session at any time or to ensure that it terminates in an unusable state, you can call the LOGOUT( ) method on the client-principal object handle (see Table 13: Client-principal object methods on page 118). This effectively clears and resets the user identity maintained by this client-principal object to blank, and the client-principal object becomes unusable. OpenEdge does not automatically logout or otherwise terminate a user login session, unless you set the LOGIN-EXPIRATION-TIMESTAMP attribute on the client-principal object handle (see Table 11: Client-principal object attributes you can set on page 113), before you seal the client-principal object.

Note: After you no longer need a client-principal object, be sure to delete it using the DELETE OBJECT statement, or you can re-initialize it using the INITIALIZE( ) method to authenticate a different user identity. For more information, see Initializing a client-principal for user authentication and SSO on page 132.

At any time during initialization and management of a user login session you can query the client-principal object attributes directly (see Table 11: Client-principal object attributes you can set on page 113 and Table 12: Client-principal object attributes are read-only on page 116) and any properties you have defined using the `GET-PROPERTY( )` method (see Table 13: Client-principal object methods on page 118). Attributes you might want to query in any user login session include the read-only `LOGIN-STATE` and `STATE-DETAIL` attributes, which reflect changes in the login session state and the usability of the associated client-principal object.

Establishing database connection identity

You can establish a database connection identity for one or more database connections in a number of ways (see Authenticating and setting user identity on page 128). Because a database connected from the command-line can be connected with a default user identity, which can provide open access to the database. Therefore, Progress Software corporation recommends that you authenticate and set the identity for all database connections entirely under ABL control using a client-principal object.

You can connect and authenticate the connection identity for all databases using the `CONNECT` statement, for which OpenEdge creates and seals a client-principal for each connection identity that you can return for other uses. However, to take full advantage of all client-principal settings, and certainly to allow your application to perform its own user authentication, you can also authenticate and set the identity for any database after it is connected using a client-principal that you create yourself.

For OpenEdge-performed user authentication and SSO for database connections, ABL supports the `SET-DB-CLIENT` function and, additionally, the `SECURITY-POLICY:SET-CLIENT( )` method. To seal a client-principal after application-performed user authentication, ABL supports the `SEAL( )` method, after which you can use `SET-DB-CLIENT` or `SET-CLIENT( )` in SSO operations to set the authenticated identity for one or more database connections.

The following sections describe some of the mechanisms that OpenEdge supports for using these ABL elements to authenticate and set database connection identity.

Built-in client support for authenticating database connection identity

OpenEdge provides built-in support for client authentication of OpenEdge database connections. If the security administrator has defined authentication-enabled domains in an OpenEdge database, your application can prompt for a user ID and password at the appropriate point to establish a database connection identity by running the standard OpenEdge startup pro2cedure, `_prostar.p`. This procedure, in turn, runs the standard ABL authentication procedure, `_login.p`, for each connected database. Your application can also reset a current connection identity, as your application requires.

The `_prostar.p` procedure also prepares `_login.p` to run appropriately in the current application environment (character or graphical) and verifies that no connected databases have the logical name `DICTDB`. This allows `_prostar.p` to assign the same alias (`DICTDB`) to each connected database before calling `_login.p`. Then, `_login.p` can authenticate access as it is called for each different database using the same database name.

This is the `_login.p` procedure.

_login.p

```
DEFINE INPUT PARAMETER viewAsDialog AS LOGICAL NO-UNDO.

{ login.i }
DEFINE VARIABLE tries     AS INTEGER NO-UNDO.
DEFINE VARIABLE lFound    AS LOGICAL NO-UNDO.
DEFINE VARIABLE hCP       AS HANDLE  NO-UNDO.
DEFINE VARIABLE setdbclnt AS LOGICAL NO-UNDO.
DEFINE VARIABLE cUserid AS LOGICAL NO-UNDO.

IF USERID("DICTDB") <> ""  THEN
    RETURN.
```

```
IF INTEGER(DBVERSION("DICTDB")) > 10 THEN DO:
    FIND FIRST  dictdb._sec-authentication-domain
            WHERE (dictdb._sec-authentication-domain._Domain-enabled = YES
              AND  dictdb._sec-authentication-domain._Domain-id > -1 AND
                   dictdb._sec-authentication-domain._Domain-type = "_oeusertable")
NO-ERROR.
       IF AVAILABLE(dictdb._sec-authentication-domain) THEN
          lFound = BUFFER dictdb._user:FIND-FIRST("WHERE dictdb._User._sql-only-user =
 FALSE") NO-ERROR.

       IF NOT lFound THEN
       DO:
          FOR EACH dictdb._sec-authentication-domain
             WHERE (dictdb._sec-authentication-domain._Domain-enabled = YES
               AND  dictdb._sec-authentication-domain._Domain-type <> "_oeusertable"
               AND  dictdb._sec-authentication-domain._Domain-id > -1 ) ,
           FIRST dictdb._sec-authentication-system OF dictdb._sec-authentication-domain

                WHERE (dictdb._sec-authentication-system._PAM-plug-in = YES ) :
                lFound = YES.
                LEAVE.
          END.
       END.

    IF lFound = FALSE THEN RETURN.
END.
ELSE IF NOT CAN-FIND(FIRST DICTDB._User) THEN
    RETURN.

CREATE CLIENT-PRINCIPAL hCP. /* create a CLIENT-PRINCIPAL only once during login*/

DO ON ENDKEY UNDO, LEAVE:

    currentdb = LDBNAME("DICTDB").

    /* reset id and password to blank in case of retry */
    ASSIGN id = ""
           domain = ""
           password = "".
     if viewAsDialog then

    do:

      DISPLAY currentdb WITH FRAME logindb_frame view-as dialog-box.

      UPDATE id password domain ok_btn cancel_btn help_btn {&WHEN_HELP}
             WITH FRAME logindb_frame view-as dialog-box.
    end.
    else do:
      DISPLAY currentdb WITH FRAME login_frame.

      UPDATE id password domain ok_btn cancel_btn help_btn {&WHEN_HELP}
             WITH FRAME login_frame.
    end.
    cUserid = id
            + if domain = "" then "" else "@" + domain.

    /* Use SET-DB-CLIENT instead of SETUSERID */
    hCP:INITIALIZE(cUserid,?,?,password).

    setdbclnt = SET-DB-CLIENT(hCP,currentdb) NO-ERROR.
    if NOT setdbclnt THEN
    DO:
        MESSAGE "Userid/Password is incorrect."
                VIEW-AS ALERT-BOX ERROR BUTTONS OK.
        IF tries > 1 THEN
            QUIT. /* only allow 3 tries*/
        tries = tries + 1.
```

```
        UNDO, RETRY.
    END.
END.

DELETE OBJECT hCP.
hCP = ?.

HIDE FRAME login_frame.
```

The _login.p procedure initializes a client-principal object by passing the fully qualified user ID and password that the user enters to the INITIALIZE( ) method and uses the ABL SET-DB-CLIENT function to authenticate the user's identity that is passed in the client-principal object. The user has three tries to enter the correct user ID and password for each database. If the user fails to do so after three tries, OpenEdge exits the user from the application. If the user ID and password combination is valid for the database, SET-DB-CLIENT establishes that user ID as the database connection ID and deletes the client-principal.

Note: Following execution of _login.p, to set the ABL session identity using the same identity as a database connection, the application can invoke the GET-DB-CLIENT function on that connection to return a copy of the client-principal. For more information on setting the ABL session identity, see Establishing ABL session identity on page 141.

The input parameter for _login.p allows it to display the authentication prompts either in a dialog box (viewAsDialog = TRUE) or in the frame of a separate window (viewAsDialog = FALSE). The _prostar.p procedure uses a separate window in graphical environments and the default window in character environments, so it always passes FALSE as an argument to _login.p.

As explained previously, the _login.p procedure only works for a database with the DICTDB alias. (By default, this alias is assigned to the first database you connect to during a session.) If you want to avoid this restriction, you can create your own procedures, based on _prostar.p and _login.p, that pass an argument with the database name.

If your application uses server authentication, you might also create your own client login procedure, for example, that replace the call to SET-DB-CLIENT in _login.p with a call to a remote login procedure on the authentication server. Typically, this remote login procedure invokes the SECURITY-POLICY:SET-CLIENT( ) method to authenticate the user identity and returns a sealed client-principal to the client, which then calls SET-CLIENT( ) to set the client session identity and possibly the connection identity for all connected application databases.

If the application does not run _prostar.p at connection time, or if the user bypasses _login.p (by pressing **END–ERROR** when prompted for the user ID and password), the user is assigned an OpenEdge default user ID. For more information on the OpenEdge default user identity, see Authenticating and managing user identity in ABL on page 104.

If you connect to a database dynamically using the CONNECT statement, you can use the User ID (–U) and Password (–P) connection parameters in the CONNECT statement, or you can use the SET-DB-CLIENT (or SETUSERID) function to reset the connection user identity after the connection is already established.

Note: While the OpenEdge default user identity can connect to a database (unless database options are set to prevent it), it cannot be used to reset an existing database connection identity.

The following i-passts.p procedure connects to the mywork database that has a list of valid users. The user initially connects to the database using the CONNECT statement with the OpenEdge default user identity. The code then enters a loop that forces the user to provide a valid user ID and password for the database user list.

i-passts.p

```
DEFINE VARIABLE passwd  AS CHARACTER NO-UNDO FORMAT "x(16) LABEL "Password".
DEFINE VARIABLE success AS LOGICAL   NO-UNDO.
DEFINE VARIABLE user-id AS CHARACTER NO-UNDO FORMAT "x(32)" LABEL "User ID".
DEFINE VARIABLE hCP     AS HANDLE    NO-UNDO.

CONNECT mywork.

DO WHILE NOT success:
  MESSAGE "Enter a user ID and password for the database mywork.".
  SET user-id passwd BLANK.

  CREATE CLIENT-PRINCIPAL hCP.
  hCP:INITIALIZE(user-id,?,?,passwd).

  IF SET-DB-CLIENT(hCP, "mywork") THEN
    success = TRUE.
  ELSE DO:
    BELL.
    MESSAGE "Invalid user ID and password; please try again.".
  END.
END.
```

Setting connection identity with application-performed user authentication

If you choose to set a database connection identity from a user identity that your application authenticates (instead of OpenEdge or an ABL callback), you must have a domain entry in the domain registry trusted by the OpenEdge database that matches the user's domain, and this domain must be configured with the built-in `_extsso` authentication system or a user-defined authentication system. In other words, the domain **cannot** be authentication-enabled.

Note: You can set the user identity for multiple database connections using a single client-principal object by configuring database options to have each database use the ABL session domain registry. In this case, the connection identity can be managed along with the ABL session identity (see Establishing ABL session identity on page 141). For more information on setting database options to use the session domain registry, see the sections on run-time domain configuration in *Introduction to Identity Management* and the online help for setting database options in the Data Administration tools.

To set a connection identity that your application authenticates, the database must already be connected with either an OpenEdge default identity or a user identity that OpenEdge has authenticated in an authentication-enabled domain. Also, any user identity that your application authenticates has no OpenEdge-supported functional identity until you specifically validate and set that identity using an SSO operation on the sealed client-principal.

To validate and set an application-authenticated database connection identity:

1. Build the session domain registry using methods of the `SECURITY-POLICY` system handle (typically done at session startup).
2. Input the required user credentials, including an indication of the security domain and a passphrase (example).
3. Create and initialize a client-principal object with appropriate values associated with the input user credentials to set the object `USER-ID`, `DOMAIN-NAME`, `SESSION-ID`, and any other attributes or application properties (using the `SET-PROPERTY( )` method.
4. Authenticate the required user credentials through the external authentication system.
5. If the authentication succeeds, proceed to step 6. If the authentication fails, invalidate the client-principal object with the `AUTHENTICATION-FAILED( )` method and exit these steps accordingly.

6. Seal the client-principal object using the domain access code configured for the security domain in the session domain registry, starting a user login session for the identity the object represents.

   Note: The authenticated identity has not yet been assigned to any resource yet.

7. Assert and validate the authenticated user identity represented by the sealed client-principal object in an SSO operation for the database connection using the ABL `SET-DB-CLIENT` function. The `SET-DB-CLIENT` function validates the client-principal object against the database trusted domain registry in order to set the corresponding database connection identity. Optionally, you can invoke `SET-DB-CLIENT` without specifying a database connection and OpenEdge validates and (if valid) sets the identity against the database trusted domain registry for each active database connection in the session.
8. Manage the rest of the user login session as usual. For more information, see Starting and maintaining user login sessions on page 134.

Thus, you might use a code fragment such as this, which starts up an application-authenticated user login session in order to set a database connection identity for a connected inventory database. The bold-faced code features the user login session management and authentication elements, as shown:

Login session for an externally authenticated database connection ID

```
DEFINE VARIABLE cAccessCode AS CHARACTER NO-UNDO.
DEFINE VARIABLE cPassword   AS CHARACTER NO-UNDO.
DEFINE VARIABLE cUserID     AS CHARACTER NO-UNDO.
DEFINE VARIABLE hAuthProc   AS HANDLE    NO-UNDO.
DEFINE VARIABLE hCP         AS HANDLE    NO-UNDO.

/* Declare external authentication functions */
FUNCTION authenticateMyUser RETURNS LOGICAL
  (INPUT cUserID AS CHARACTER, INPUT cPassword AS CHARACTER) IN hAuthProc.
FUNCTION getAccessCode RETURNS CHARACTER INhAuthProc.

/* Get user ID/password and instantiate external authentication object,
   which also provides the authentication domain access code */
...
cUserID = ...
cPassword = ...
RUN MyAuthentication.p PERSISTENT SET hAuthProc.
cAccessCode = getAccessCode().

/* Create and initialize a client-principal for use with an authentication
   domain in the database domain registry */
CREATE CLIENT-PRINCIPLE hCp.
ASSIGN
  hCP:SESSION-ID  = "Inventory:" + cUserID + STRING(NOW)
  hCP:USER-ID     = cUserID
  hCP:DOMAIN-NAME = "DatabaseUsers".

/* Authenticate and login database connection ID
   session for the single database connection */
If authenticateMyUser(cUserID, cPassword) THEN DO:
  hCP:SEAL(cAccessCode).
  IF NOT SET-DB-CLIENT(hCP) THEN DO:
    hCP:LOGOUT().
    cRetVal = "User ID not valid DB connection ID".
  END.
END.
ELSE DO:
  hCP:AUTHENTICATION-FAILED("User not authenticated.").
  cRetVal = hCP:LOGIN-STATE + ": " + hCP:STATE-DETAIL.
END.
/* Handle results of authentication */
IF hCP:LOGIN-STATE = "LOGIN" THEN DO: /* Do Inventory stuff... */
  ...
  /* Clear the database connection ID to end the login session */
  hCP:LOGOUT().
  DELETE OBJECT hCP.
END.
ELSE DO: /* Exit with failure message */
  RETURN cRetVal.
  DELETE OBJECT hCP.
END.
```

In this fragment, the domain access code used by the application is maintained and returned by the same authentication procedure (`MyAuthentication.p`) that authenticates the user identity. This value, returned using the `getAccessCode` user-defined function, must match the access code configured for the authentication domain entry in the database trusted domain registry.

Also in this code fragment, the setting of the `SESSION-ID` attribute, which includes the value of `STRING(NOW)`, while valid is not necessarily unique. If you enable auditing for the application, where unique references are more important, use the `GENERATE-UUID` function (or for a multi-tier application, the client context ID) to set the `SESSION-ID` attribute. For more information, see Creating and managing unique object identities on page 154.

Establishing ABL session identity

An ABL session identity can serve a variety of functions, including as a:

- Single session-wide and database-independent identity for authorizing application-defined features
- Single session-wide database connection identity for all database connections in an ABL session (see Establishing database connection identity on page 135)
- As a session-wide auditing identity (see Establishing auditing identity on page 154)
- Single application-wide identity for all ABL sessions and database connections in a multi-tier application (see Establishing and managing identity for multi-tier applications on page 145)

If you do not explicitly set it, the ABL session identity is undefined, with the client-principal handle value returned by the `SECURITY-POLICY:GET-CLIENT( )` method set to the Unknown value (?). You can also clear an existing ABL session identity, which resets the effective session identity to the unknown identity.

To set an ABL session identity, you must build a session domain registry for the session at run time in order to assert and validate an authenticated user identity as a valid ABL session identity for SSO, and optionally as a database connection identity. For more information, see Setting up and using domain registries on page 123.

Setting the ABL session identity

Many ABL applications authenticate a user identity externally. For user identities that are asserted in an authentication-enabled domain, you can authenticate and set a session identity directly from an unsealed client-principal using the `SECURITY-POLICY:SET-CLIENT( )` method. OpenEdge then performs the user authentication, (if successful) seals the client-principal, and sets the session identity, all in a single operation.

Using a client-principal object with the session domain registry and appropriate database configuration options, you can set any authenticated user identity as the ABL session identity, which OpenEdge can then recognize for use with various OpenEdge features. For example:

- If you select the database option (using Data Administration tools), **Use Application Domain Registry**, OpenEdge uses the current session registry (instead of the local database domain registry) for authenticating and validating any user identity assigned to the database connection, including the ABL session identity. In this way, the database can authorize access to its tables and fields based on the effective ABL session identity. If all connected OpenEdge databases are configured to use the application domain registry, you can efficiently set all of their database connection identities to this single ABL session identity. Thus, you can use the ABL session identity to synchronize all user identities for the entire ABL session.
- If you select the database option, **Use Application User Id for Auditing**, the database records the ABL session user ID as the auditing ID in its audit event records.

  Note: Set the database option, **Use Application User Id for Auditing**, **only** if you are individually managing the database connection identity for each database (see Establishing database connection identity on page 135). If you synchronize all database connection identities with the ABL session identity, all connected databases use the ABL session user ID as the auditing ID by default.

- You can set a single ABL session identity and use a copy of the sealed client-principal to transport this user identity between OpenEdge sessions of a distributed application to maintain a consistent identity among them (see Establishing and managing identity for multi-tier applications on page 145).

If you want to set an ABL session identity from a user identity that you authenticate externally, you must have a corresponding domain entry registered in the session domain registry. As described in an earlier section (see Setting up and using domain registries on page 123), you can build this registry within the ABL session at run time entirely from a local database domain registry or you can build it one domain entry at a time directly from ABL code. It is generally easier and more secure to load the session domain registry from a local database domain registry.

Any user ID that you authenticate with an OpenEdge-supported authentication system has an OpenEdge-supported functional identity after you validate that identity against the corresponding authentication domain. This you must do using a client-principal object.

To authenticate an ABL session identity:

1. Build the session domain registry using methods of the `SECURITY-POLICY` system handle (typically done at session startup).
2. Input the required user credentials, including an indication of the security domain and a passphrase (example).
3. Create and initialize a client-principal object with appropriate values associated with the input user credentials to set the object `USER-ID`, `DOMAIN-NAME`, `SESSION-ID`, and any other attributes or application properties (using the `SET-PROPERTY( )` method.)
4. If the user's domain is authentication-enabled, authenticate the user's identity asserted with the client-principal using the `SECURITY-POLICY:SET-CLIENT( )` method. Otherwise, proceed to step 6.
5. If the authentication succeeds, the asserted identity is now the session identity. If the authentication fails, the client-principal object is set to the FAILED state. Exit these steps accordingly.
6. If the user's domain supports application-performed authentication, authenticate the required user credentials using the specified external authentication system.
7. If the authentication succeeds, proceed to Step 8. If the authentication fails, invalidate the client-principal object with the `AUTHENTICATION-FAILED( )` method and exit these steps accordingly.
8. Seal the client-principal object using the domain access code configured for the security domain in the session domain registry, starting a user login session for the identity the object represents.

   Note: The authenticated identity has not yet been assigned to any resource yet.

9. Assert and validate the user identity represented by the client-principal object in an SSO operation for the database connection using the ABL `SET-DB-CLIENT` function. The `SET-DB-CLIENT` function validates the client-principal object against the database trusted domain registry in order to set the corresponding database connection identity. Optionally, you can invoke `SET-DB-CLIENT` without specifying a database connection and OpenEdge validates and (if valid) sets the identity against the database trusted domain registry for each active database connection in the session.
10. Assert and validate the authenticated user identity represented by the sealed client-principal object in SSO operation for the ABL session using the `SECURITY-POLICY:SET-CLIENT( )` method. The `SET-CLIENT( )` method validates the client-principal object against the session domain registry in order to set the ABL session identity.

    In addition, for any existing database connections whose identities have not **already** been set using the `SETUSERID` or `SET-DB-CLIENT` functions, the `SET-CLIENT( )` method also attempts to validate and set the connection identity in an SSO operation against the database trusted domain registry for each of these databases. This setting can always be overridden using the `SETUSERID` or `SET-DB-CLIENT` functions. However, if options for a given database are set to use the application domain registry, both `SETUSERID` and `SET-DB-CLIENT` use the session registry to validate and set the connection identity for that database.

Note: If the session domain registry is not initialized or remains unlocked, `SET-CLIENT( )` does not validate an ABL session identity, but does validate the client-principal object and set database connection identities similar to the `SET-DB-CLIENT` function called without specifying a database connection (see Setting connection identity with application-performed user authentication on page 138).

11. Manage the rest of the user login session as usual (see Starting and maintaining user login sessions on page 134).

Thus, you might use a code fragment such as the following, which starts up an application-authenticated user login session in order to set an ABL session identity for an inventory application that accesses a single database (already connected). The bold-faced code features the user login session management and authentication elements, as shown:

Login session for an externally authenticated OpenEdge session ID

```
DEFINE VARIABLE cAccessCode AS CHARACTER NO-UNDO.
DEFINE VARIABLE cPassword   AS CHARACTER NO-UNDO.
DEFINE VARIABLE cUserID     AS CHARACTER NO-UNDO.
DEFINE VARIABLE hAuthProc   AS HANDLE    NO-UNDO.
DEFINE VARIABLE hCP         AS HANDLE    NO-UNDO.

/* Declare external authentication functions */
FUNCTION authenticateMyUser RETURNS LOGICAL
  (INPUT cUserID AS CHARACTER, INPUT cPassword AS CHARACTER) IN hAuthProc.
FUNCTION getAccessCode RETURNS CHARACTER IN hAuthProc.

/* Build the application trusted domain registry from the database domain
   registry available for the single database connection */
SECURITY-POLICY:LOAD-DOMAINS(1).

/* Get user ID/password and instantiate external authentication object,
   which also provides the authentication domain access code */
...
cUserID = ...
cPassword = ...
RUN MyAuthentication.p PERSISTENT SET hAuthProc.
cAccessCode = getAccessCode().

/* Create and initialize a client-principal for use with an authentication
   domain in the application domain registry */
CREATE CLIENT-PRINCIPLE hCp.
ASSIGN
  hCP:SESSION-ID  = "Inventory:" + cUserID + STRING(NOW)
  hCP:USER-ID     = cUserID
  hCP:DOMAIN-NAME = "InventoryApp".

/* Authenticate and login OpenEdge session ID for the application */
If authenticateMyUser(cUserID, cPassword) THEN DO:
  hCP:SEAL(cAccessCode).
  /* Assert and validate ID against application domain registry */
  IF NOT SECURITY-POLICY:SET-CLIENT(hCP) THEN DO:
    hCP:LOGOUT().
    cRetVal = "User ID not valid OpenEdge session ID".
  END.
END.
ELSE DO:
  hCP:AUTHENTICATION-FAILED("User not authenticated.").
  cRetVal = hCP:LOGIN-STATE + ": " + hCP:STATE-DETAIL.
END.
/* Handle results of authentication */
IF hCP:LOGIN-STATE = "LOGIN" THEN DO: /* Do Inventory stuff... */
  ...
  /* Clear the OpenEdge session ID to end the login session */
  hCP:LOGOUT().
  DELETE OBJECT hCP.
END.
ELSE DO: /* Exit with failure message */
  RETURN cRetVal.
  DELETE OBJECT hCP.
END.
```

In this fragment, the domain access code used by the application is maintained and returned by the same authentication procedure (`MyAuthentication.p`) that authenticates the user identity. This value, returned using the `getAccessCode` user-defined function, must match the access code configured for the authentication domain entry in the session domain registry.

Also in this code fragment, the setting of the SESSION-ID attribute, which includes the value of STRING(NOW), while valid is not necessarily unique. If you enable auditing for the application, where unique references are more important, use the GENERATE-UUID function (or for a multi-tier application, the client context ID) to set the SESSION-ID attribute. For more information, see Creating and managing unique object identities on page 154.

Note: If no database connection identity is set for the connected database, the SET-CLIENT( ) method in this fragment also sets the database connection identity, by default, using the database's own trusted domain registry. If database options are set to trust the application domain registry, SET-CLIENT( ) uses the session domain registry (which in this example has the same content as the local database domain registry) to set the database connection identity. Also, if you replace SECURITY:SET-CLIENT(hCP) with SET-DB-CLIENT(hCP) in this code, the entire fragment only sets and manages the database connection identity, and the OpenEdge session identity is undefined.

Establishing and managing identity for multi-tier applications

An application user identity exists essentially to specify a single user identity that is shared between client and AppServer sessions of a multi-tier application. The application user identity typically originates from the client and is used to set the OpenEdge session identity in each AppServer session. It might also be used for synchronizing database connection identity in each AppServer session.

In multi-tier applications, managing application user identity requires resolving these key issues:

- How to securely authenticate and transport the application user identity between AppServer sessions and a given client session
- How to share a single login for an application user identity across multiple requests to an AppServer running in stateless or state-free operating mode

Therefore, when running in stateless or state-free operating mode, an AppServer must:

- Authenticate the user's identity once per client connection (stateless) or once per user login (state-free)
- Share copies of a single client-principal object using a common client context storage to assert the user's identity on each agent involved in a given client's requests
- Logout and destroy all copies of the client-principal object when the client disconnects (stateless) or logs out (state-free)

In OpenEdge, the basic mechanism for managing user identities is the client-principal object. Typically, in a multi-tier application, ABL code on an AppServer (authentication server) creates and maintains the client-principal object on behalf of the client and exchanges identity information, based on the operating mode, as described in the following table.

Table 15: Identity management for multi-tier applications

In this operating mode...	The AppServer...
State-aware or state-reset	Maintains a single client connection to a given agent. So, the agent only needs to assert and maintain the user's identity using a single client-principal object during the entire connection. The agent can then remove the client-principal object from session context when the client disconnects.

In this operating mode...	The AppServer...
Stateless	Maintains multiple client connections to the broker, which distributes client requests to any available agent. Because the broker maintains client connections, a shared client-principal object can be identified using the `SERVER-CONNECTION-ID` on the `SESSION` system handle. However, this value is only unique within an AppServer session. For auditing across multiple AppServer sessions, you might use a universal unique identifier (UUID) generated by the ABL `GENERATE-UUID` function to uniquely identify any recorded user login sessions, and use the `SERVER-CONNECTION-ID` on the `SESSION` system handle to key access to the client-principal object in client context storage. Alternatively, you can use the `ClientContextId` property on the `Progress.Lang.OERequestInfo` object as described for state-free operating mode for both auditing and to key access to client context storage.
State-free	Maintains no client connections. The broker distributes client requests as they arrive to any available agent. Because the broker maintains no client connections, the shared client-principal object must be identified based on a unique identifier, which you can generate using the ABL `GENERATE-UUID` function or which you can obtain and set as the value of the `ClientContextId` property on an `Progress.Lang.OERequestInfo` object using the following handle attributes, which return object references to instances appropriate for either sending the property value to the AppServer in a client request or returning the property value to the client in a server response: • **AppServer client** — The `REQUEST-INFO` and `RESPONSE-INFO` attributes on the server object handle for a given AppServer • **AppServer agent** — The `CURRENT-REQUEST-INFO` and `CURRENT-RESPONSE-INFO` attributes on the `SESSION` system handle of the agent that is servicing a given client request This built-in support for different client and server instances of a `Progress.Lang.OERequestInfo` object conveniently allows the passing of a single `ClientContextId` property value on behalf of a single user identity to any level of a multi-tier application, where AppServers send client requests, in turn, to other AppServers. `Progress.Lang.OERequestInfo` also includes the `GetClientPrincipal( )`and `SetClientPrincipal( )`, for accessing the client-principal object. For example, in an application that uses server authentication, the authenticating AppServer can provide this value to both initialize and store each client-principal object that it authenticates, seals, and exports back to the client in response to its initial login request. The client can then return the same unique value with each request to an application AppServer in the same login session. The responding AppServer agent can then use the value as a key to import the shared client-principal object from client context storage to set its application identity before servicing the client request, and so on.

Initializing a client-principal object for a multi-tier application

The code examples in this section show two approaches to initializing the client-principal object for a multi-tier application. Both examples test for the application server operating mode and both add the sealed client principal to a database that stores session context. The examples differ in the way the client-principal object is accessed.

The following code fragment shows the use of the `GetClientPrincipal ( )` method to access an unsealed client-principal object. The client-principal object is created in a `Progress.Lang.OERequestInfo` instance. `OERequestInfo` is automatically initialized when a `CREATE SERVER` statement executes on an ABL client and can be referenced using the attributes of the `SESSION` system handle.

The following code fragment also shows the use of the `SetClientPrincipal ( )` method to store a deep copy (clone) of the client principal in the `OERequestInfo` instance after either the success or failure of authentication.

Client principal initialization using GetClientPrincipal ( ) and SetClientPrincipal ( ) methods

```
DEFINE VARIABLE hCP AS HANDLE NO-ERROR.

/* Get unsealed Client Principal from client and create a deep copy.*/

hCP:SESSION:CURRENT-REQUEST-INFO:GetClientPrincipal().

IF (SESSION:SERVER-OPERATING-MODE = "State-free" OR
    SESSION:SERVER-OPERATING-MODE = "Stateless") THEN DO:
    hCP:SESSION-ID = SESSION:CURRENT-REQUEST-INFO:ClientContextId.

END. /* ELSE Other session-managed operating modes don't need it */

/* Initialize audit and other context info */
m_hCP:AUDIT-EVENT-CONTEXT = m_hCP:USER-ID + "@" + hCP:DOMAIN-NAME.
m_hCP:CLIENT-TTY = SESSION:CLIENT-TYPE + "." + SESSION:DISPLAY-TYPE.

/* Authenticate and seal client-principal */
IF SECURITY-POLICY:SET-CLIENT(hCP) THEN /* Authn succeeded */
  SESSION:CURRENT-RESPONSE-INFO:SetClientPrincipal(hcp). /* Return sealed Client
Principal*/
  IF (SESSION:SERVER-OPERATING-MODE = "State-free" OR
      SESSION:SERVER-OPERATING-MODE = "State-aware") THEN DO:
    /* Add sealed Client Principal to client context store */
    ASSIGN
      ContextDB.ContextID = hCP:SESSION-ID /* Add key to context store */
      ContextDB.ContextC-P = hcp:EXPORT-PRINCIPAL(). /* Add sealed Client Principal as
 RAW value to context store */
  END.
ELSE DO: /* If authn fails, return failed C-P to client*/
  SESSION:CURRENT-RESPONSE-INFO:SetClientPrincipal(hcp).
END.
DELETE OBJECT hCP.
```

The next code fragment shows the use of the `IMPORT-PRINCIPAL ( )` and `EXPORT-PRINCIPAL ( )` methods for initializing the client-principal object. Using these methods requires that the client-principal object is explicitly created in the code, rather than referenced as an attribute of the `SESSION` system handle, as in the previous example. Also note, that in this case the client-principal object is passed as a parameter rather than referenced as an attribute.

Client-principal initialization using the IMPORT-PRINCIPAL ( ) and EXPORT-PRINCIPAL ( ) methods

```
DEFINE INPUT  PARAMETER prCPIn  AS RAW NO-ERROR. /* Unsealed C-P to authn */
DEFINE OUTPUT PARAMETER prCPOut AS RAW NO-ERROR. /* Sealed C-P to return */

DEFINE VARIABLE m_hCP AS HANDLE NO-ERROR.

CREATE CLIENT-PRINCIPAL m_hCP.
m_hCP:IMPORT-PRINCIPAL(prCPIn). /* Import unsealed C-P from client */

IF (SESSION:SERVER-OPERATING-MODE = "State-free" OR
    SESSION:SERVER-OPERATING-MODE = "Stateless") THEN DO:
  m_hCP:SESSION-ID = SESSION:CURRENT-REQUEST-INFO:ClientContextId.

END. /* ELSE Other session-managed operating modes don't need it */

/* Initialize audit and other context info */
m_hCP:AUDIT-EVENT-CONTEXT = m_hCP:USER-ID + "@" + m_hCP:DOMAIN-NAME.
m_hCP:CLIENT-TTY = SESSION:CLIENT-TYPE + "." + SESSION:DISPLAY-TYPE.

/* Authenticate and seal client-principal */
IF SECURITY-POLICY:SET-CLIENT(m_hCP) THEN /* Authn succeeded */
  prCPOut = m_hCP:EXPORT-PRINCIPAL( ). /* Export sealed C-P to client */
  IF (SESSION:SERVER-OPERATING-MODE = "State-free" OR
      SESSION:SERVER-OPERATING-MODE = "State-aware") THEN DO:
    /* Add sealed C-P to client context store */
    ASSIGN
      ContextDB.ContextID = m_hCP:SESSION-ID /* Add key to context store */
      ContextDB.ContextC-P = prCPOut. /* Add exported C-P */
  END.
ELSE DO: /* Authn failed */
  prCPOut = m_hCP:EXPORT-PRINCIPAL( ). /* Export failed C-P to client */
END.

DELETE OBJECT m_hCP.
```

For the state-free and stateless operating modes, both code fragments set the SESSION-ID attribute to the ClientContextId property value obtained from the OERequestInfo object returned by the CURRENT-REQUEST-INFO attribute on the SESSION system handle. This property is automatically initialized with a GUID value when the client initially creates the server handle to access the authentication server. This ClientContextId property value, as well as the sealed client-principal set on the CURRENT-RESPONSE-INFO handle in the first example, remain unchanged during all subsequent requests from the client to the application server (using the same server handle) unless either the client or the application server change them.

A stateless application service can also identify each request by the client connection and key a context store using the SERVER-CONNECTION-ID attribute on the SESSION system handle. However, it also needs a unique value to set the SESSION-ID attribute in the client-principal for auditing purposes. Because the same ClientContextId property value is both globally unique and can be easily passed between sessions, the management of stateless context can be simplified by using this one value for both purposes instead of creating a separate unique value for auditing with the ABL GENERATE-UUID function. This one inter-session value ensures that a unique user login session auditing record is recorded across all possible application server sessions and can also be used to key a context store for the login session.

Both stateless and state-free application services must manage user context in order to export, store, retrieve, and import different client-principal objects between client requests. In releases prior to OpenEdge 11.0, but they must do it differently according to their respective session-managed and session-free behavior. With Release 11.0, application services running in either operating mode can take advantage of the same ClientContextId property mechanism.

However, the SESSION:SERVER-CONNECTION-ID attribute and GENERATE-UUID function mechanisms continue to work for stateless operating mode. The following sections provide more detail using different examples on how to manage identity for both stateless and state-free application services.

Managing user identities for a stateless application service

In a stateless application service, the process of managing user identities is somewhat simplified by the existence of client connections. Following is a typical procedure for managing stateless user identities authenticated using application-performed user authentication and the `SESSION:SERVER-CONNECTION-ID` attribute and the `GENERATE-UUID` function.

Note: A stateless application service can also manage user identities using the same client context ID mechanism that works for state-free application services. For more detailed information, see Managing user identities for a state-free application service on page 151.

To manage users identities for a stateless application service, you can do the following:

1. In the AppServer's configured startup procedure, build any required session domain registry and remove any client-principal objects from a previous user context left-over from any abnormal session termination.
2. In the application service's login procedure:
   a. Initialize and seal the client-principal object using a UUID (to uniquely set any auditing context) as the value for the `SESSION-ID` attribute on the client-principal object.
   b. Export the client-principal object.
   c. Store the exported `RAW` value of the client-principal object in a context database keyed on the `SESSION:SERVER-CONNECTION-ID` attribute value.

   For example:

```
DEFINE SHARED VARIABLE cAccessCode AS CHARACTER NO-UNDO.
DEFINE SHARED VARIABLE hCP         AS HANDLE    NO-UNDO.

DEFINE VARIABLE cSessionID AS CHARACTER NO-UNDO.
DEFINE VARIABLE rCP        AS RAW       NO-UNDO.

CREATE CLIENT-PRINCIPAL hCP.

/* Initialize, authenticate, seal, and login client-principal object */
...
cSessionID     = BASE64-ENCODE(GENERATE-UUID).
hCP:SESSION-ID = SUBSTRING(cSessionID, 1, 22)
hCP:SEAL(cAccessCode).

/* Export client-princpal object */
rCP = hCP:EXPORT-PRINCIPAL().

/* Store exported client-principal by the connection ID */
CREATE UserContext NO-ERROR.
ASSIGN UserContext.Principal-key = SESSION:SERVER-CONNECTION-ID
       UserContext.Principal     = rCP NO-ERROR.
...
```

3. In the AppServer's configured activation procedure:
   a. Lookup the exported client-principal object in the context database using the `SESSION:SERVER-CONNECTION-ID` value as the lookup key
   b. Import the client principal object
   c. Validate the imported client-principal seal, and use the validated client principal object to set the user identity

For example:

```
DEFINE SHARED VARIABLE hCP AS HANDLE NO-UNDO.

DEFINE VARIABLE rCP AS RAW NO-UNDO.

CREATE CLIENT-PRINCIPAL hCP.

/* Lookup client-principal object in context database */
FIND UserContext
  WHERE Principal-key = SESSION:SERVER-CONNECTION-ID NO-ERROR.
rCP = UserContext.Principal.

/* Import client-princpal object, validate, and set user identity */
hCP:IMPORT-PRINCIPAL(rCP).
IF SECURITY-POLICY:SET-CLIENT(hCP) THEN /* Identity is valid */
  ...
ELSE /* Identity is not valid */
  ...
```

4. In all non-login procedures for the application service, verify that the user identity is valid from activation procedure, and if it is, perform the specified application service task. For example:

```
DEFINE SHARED VARIABLE cAccessCode AS CHARACTER NO-UNDO.
DEFINE SHARED VARIABLE hCP         AS HANDLE    NO-UNDO.

IF hCP:VALIDATE-SEAL(cAccessCode) THEN /* Identity is valid */
    ... /* Perform application service task */

ELSE /* Identity is not valid */
    ...
```

5. In the AppServer's configured deactivation procedure, clear the ABL session identity for the next user. For example:

```
DEFINE SHARED VARIABLE hCP AS HANDLE NO-UNDO.

hCP:LOGOUT().
DELETE OBJECT hCP.
...
```

6. In the application service's logout procedure:

a. Lookup the exported client-principal object in the context database using the SESSION:SERVER-CONNECTION-ID value as the lookup key

b. Remove the exported client-principal object from the context database, for example:

```
/* Lookup client-principal object and remove from context database */
FIND UserContext
  WHERE Principal-key = SESSION:SERVER-CONNECTION-ID NO-ERROR.
DELETE UserContext NO-ERROR.
...
```

7. In the AppServer's configured shutdown procedure, empty the user context database, for example:

```
/* Empty the context database */
FOR EACH UserContext:
  DELETE UserContext NO-ERROR.
END.
...
```

Managing user identities for a state-free application service

In a state-free application service, the process of managing user identities is somewhat more complicated by the lack of any client connections. This requires the application service to manage user identities independently of any physical client identity and to provide a means to exchange the client's identity with the application service on every client request. Note that for a state-free AppServer, there are no configurable activation and deactivation procedures. So, all the work of establishing and clearing user identities for a given client request must be done for each and every service call by the application service itself. However, this process is simplified using the client context ID mechanism that is available for both state-free and stateless application services. Following is a typical procedure for managing state-free user identities authenticated using application-performed user authentication together with this client context ID mechanism.

To manage users identities for a state-free application service:

1. In the AppServer's configured startup procedure, build any required session domain registry and remove any client-principal objects from a previous user context left-over from any abnormal session termination.
2. In the application service's login procedure:

   a. Initialize and seal the client-principal object, including setting a unique value for the SESSION-ID attribute from the ClientContextId property on the Progress.Lang.OERequestInfo object available through the SESSION:CURRENT-REQUEST-INFO attribute.

   b. Export the client-principal object.

   c. Store the exported RAW value of the client-principal object in a context database keyed on the SESSION-ID attribute value.

   d. Return the SESSION-ID attribute value (client context ID) as an output parameter to the client. The client will then call all non-login procedures of the application service by assigning this client context ID to the ClientContextID property on the Progress.Lang.OERequstInfo object that is available through the REQUEST-INFO attribute on the server object handle for the application service.

For example:

```
DEFINE OUTPUT PARAMETER cSessionID AS CHARACTER NO-UNDO.

DEFINE SHARED VARIABLE hCP AS HANDLE NO-UNDO.

DEFINE VARIABLE cAccessCode AS CHARACTER NO-UNDO.
DEFINE VARIABLE rCP         AS RAW       NO-UNDO.

CREATE CLIENT-PRINCIPAL hCP.

/* Initialize, authenticate, seal, and login client-principal object */
...
hCP:SESSION-ID = SESSION:CURRENT-REQUEST-INFO:ClientContextId.
hCP:SEAL(cAccessCode).

/* Export client-princpal object */
rCP = hCP:EXPORT-PRINCIPAL().

/* Store exported client-principal */
CREATE UserContext NO-ERROR.
ASSIGN UserContext.Principal-key = hCP:SESSION-ID
       UserContext.Principal     = rCP NO-ERROR.

/* Return the client context ID value to client */
cSessionID = hCP:SESSION-ID.
...
```

3. In all non-login procedures for the application service:
   a. Retrieve the client context ID from the `ClientContextID` property on the `Progress.Lang.OERequstInfo` object that is available through the `SESSION:CURRENT-REQUEST-INFO` attribute.
   b. Lookup the exported client-principal object in the context database using the security token value as the lookup key.
   c. Import the client principal object.
   d. Validate the imported client-principal seal, and use the validated client principal object to set the user identity.
   e. Perform the application service task.
   f. Clear the ABL session identity for the next user.

For example:

```
DEFINE        VARIABLE cSessionID AS CHARACTER NO-UNDO.
DEFINE        VARIABLE rCP AS RAW    NO-UNDO.
DEFINE SHARED VARIABLE hCP AS HANDLE NO-UNDO.

cSessionID = SESSION:CURRENT-REQUEST-INFO:ClientContextId.

CREATE CLIENT-PRINCIPAL hCP.

/* Lookup client-principal object in context database */
FIND UserContext WHERE Principal-key = cSessionID NO-ERROR.
rCP = UserContext.Principal.

/* Import client-princpal object, validate, and set user identity */
hCP:IMPORT-PRINCIPAL(rCP).
IF SECURITY-POLICY:SET-CLIENT(hCP) THEN /* Identity is valid */
  ... /* Perform application service task */

ELSE /* Identity is not valid */
  ...

/* Clear user identity for next user */
hCP:LOGOUT().
DELETE OBJECT hCP.
...
```

4. In the application service's logout procedure:
   a. Retrieve the client context ID from the `ClientContextID` property on the `Progress.Lang.OERequstInfo` object that is available through the `SESSION:CURRENT-REQUEST-INFO` attribute.
   b. Lookup the exported client-principal object in the context database using the security token value as the lookup key.
   c. Remove the exported client-principal object from the context database.

   For example:

```
DEFINE VARIABLE cSessionID AS CHARACTER NO-UNDO.

cSessionID = SESSION:CURRENT-REQUEST-INFO:ClientContextId.

/* Lookup client-principal object and remove from context database */
FIND UserContext
  WHERE Principal-key = cSessionID NO-ERROR.
DELETE UserContext NO-ERROR.
...
```

5. In the AppServer's configured shutdown procedure, empty the user context database. For example:

```
/* Empty the context database */
FOR EACH UserContext:
  DELETE UserContext NO-ERROR.
END.
...
```

Establishing auditing identity

By default, OpenEdge uses the database connection identity as the auditing identity. As described in earlier sections, OpenEdge can otherwise take the auditing identity for a given database from the ABL session identity, depending on a setting in database auditing options. No matter what source you use for the auditing identity, you might note the following points:

- The auditing identity only provides the value recorded as the user ID in audit event records and has no effect on database authorization, either for table and field access or for auditing privileges. The effective database connection identity for each database always controls database access and auditing privileges.
- If you configure the database option to use the application domain registry, this causes both the ABL `SET-DB-CLIENT` function and the `SECURITY-POLICY:SET-CLIENT( )` method to assert and validate the database connection identity using the session domain registry (instead of using the local database domain registry). If you use `SET-CLIENT( )` with this database option to set the database connection identity, which also asserts and sets the ABL session identity, the effective auditing identity is identical to the ABL session identity, regardless of any database auditing options that you set.

For more information on managing the auditing ID within an OpenEdge session, see Auditing on page 241.

Creating and managing unique object identities

It is often helpful, especially for managing user identities, to obtain a value that is guaranteed to be unique. Unique values are useful anywhere you need to reference some object or data that is guaranteed not to conflict with any other of its type. A common use for unique values is to generate unique session identifiers for client-principal objects, so user login sessions can be recorded uniquely, especially for auditing purposes. Another use is to uniquely store exported client-principal objects for later retrieval to set user identities for state-free AppServer sessions (see Establishing and managing identity for multi-tier applications on page 145).

OpenEdge provides two ABL functions that work specifically with unique values, as described in the following table.

Table 16: Unique value functions

This ABL function...	Returns this value...
`GENERATE-UUID`	A 16-byte `RAW` value that represents a universally unique identifier (UUID). A UUID is guaranteed to be unique for all practical time and space.
`GUID( [` *`UUID`* `] )`	A `CHARACTER` value that represents a globally unique identifier (GUID). A GUID is UUID converted to a 36-character string value consisting of 32 hexadecimal digits formatted with 4 hyphens in a standard fashion suitable for display. If you do not specify an argument, the function generates a UUID and returns the GUID for it. If you specify a UUID, which must be 16-byte `RAW` value, the function converts the UUID argument to a GUID.

When storing unique values to use in database indexes, you need to store them in an efficient character-string format. OpenEdge provides ABL functions for encoding and decoding `RAW` (binary) data as character values for use in indexes, or other purposes where you need character storage for such a value. These functions support two different character formats for encoding and decoding `RAW` data:

- Base64
- Hexidecimal

The following table describes these functions.

Table 17: Data encoding/decoding functions

This ABL function...	Returns this value...
`BASE64-ENCODE( expression )`	A `LONGCHAR` value consisting of a Base64 representation of the `RAW` or `MEMPTR` value passed as *`expression`*
`BASE64-DECODE( expression )`	A `MEMPTR` value that represents the `CHARACTER` or `LONGCHAR` value containing a Base64 string passed as *`expression`*
`HEX-ENCODE( expression )`	A `CHARACTER` value consisting of a hexadecimal representation (an even number of the digits 0 through 9 and A through F) of the `RAW` value passed as *`expression`*
`HEX-DECODE( expression )`	A `RAW` value that represents the `CHARACTER` value containing an even number of hexadecimal digits (0 through 9 and A through F) passed as *`expression`*

For character indexes, the `BASE64-ENCODE` function works especially well with UUID values. You can convert the 16-byte UUID `RAW` value to a 24-character Base64 string and remove the two trailing pad characters for a final result containing 22 characters. For example:

```
DEFINE VARIABLE hCP AS HANDLE NO-UNDO.
CREATE CLIENT-PRINCIPAL hCP.
hCP:SESSION-ID = SUBSTRING(BASE64-ENCODE(GENERATE-UUID), 1, 22).
```

For examples of other uses for these encoding and decoding functions, see Managing and transporting crypto data on page 186.

Authorizing access to procedures and database resources

To ensure that only authorized users can access certain features in your application, you can provide run-time authorization to check the user ID of any user attempting to run a particular procedure or to access database tables and fields. ABL provides a mechanism, the `CAN-DO` function, which allows you to validate a list of user ID patterns against one of the following:

- The user ID for a given database connection (database connection ID)
- An application specified user ID

You can use the success of this validation to determine if a given procedure can be run.

You can also check the authorization (configured in a given OpenEdge RDBMS) for users to access specific database tables and fields, based on the current database connection ID.

The following sections describe:

- Authorizing users from within a given procedure on page 156

- Defining activities-based user authorization on page 159
- Authorizing user access to tables and fields on page 165

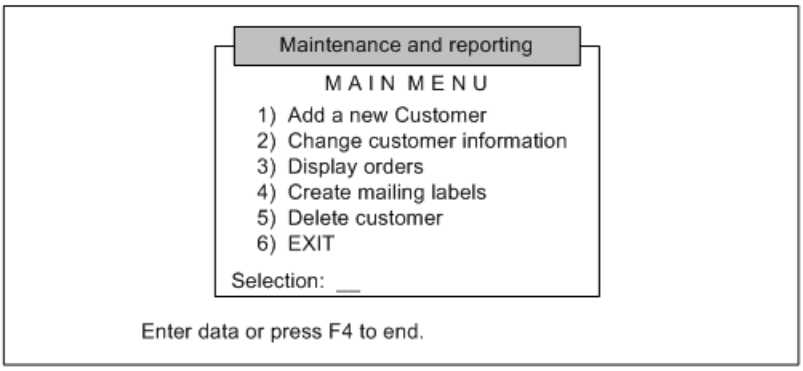
Authorizing users from within a given procedure

This sections shows some examples of procedures that you can use to check for user IDs in order to run the given procedure. The `i-csmnu3.p` procedure uses `_prostar.p`.

i-csmnu3.p

```
DEFINE VARIABLE selection AS INTEGER NO-UNDO FORMAT "9".
RUN _prostar.p.

REPEAT:
  FORM  SKIP(2)    "    M A I N M E N U"
      SKIP(1)    "  1)  Add a new customer"
      SKIP(1)    "  2)  Change customer Information"
      SKIP(1)    "  3)  Display orders"
      SKIP(1)    "  4)  Create mailing labels"
      SKIP(1)    "  5)  Delete a customer"
      SKIP(1)    "  6)  EXIT"
  WITH CENTERED TITLE "Maintenance and Reporting".
  UPDATE SKIP(2) SPACE(1) selection AUTO-RETURN
    WITH SIDE LABELS.
  HIDE.
       IF selection EQ 1 THEN RUN i-adcust.p.
  ELSE IF selection EQ 2 THEN RUN i-chcust.p.
  ELSE IF selection EQ 3 THEN RUN i-itlist.p.
  ELSE IF selection EQ 4 THEN RUN i-rept6.p.
  ELSE IF selection EQ 5 THEN RUN i-delcus.p.
  ELSE IF selection EQ 6 THEN QUIT.
  ELSE MESSAGE "Incorrect selection - please try again".
END.
```

This procedure defines user access by first running the `_prostar.p` procedure before displaying the following main menu in a character environment:



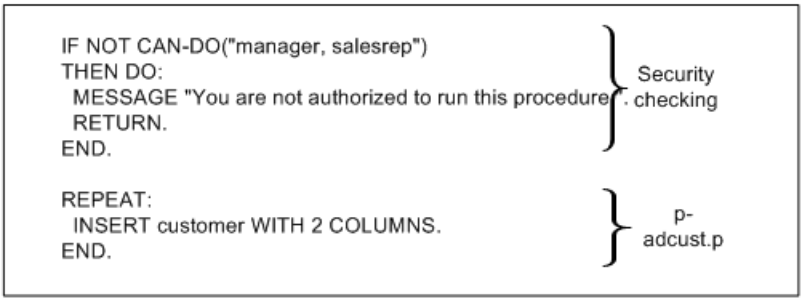
You can define, on a per procedure basis, the individuals who can run each of the Maintenance and Reporting menu procedures. Use the `CAN-DO` function to check the user ID(s) established by `_prostar.p`. The `i-adcust.p` procedure allows you to enter Customer information.

i-adcust.p

```
REPEAT:
  INSERT Customer WITH 2 COLUMNS.
END.
```

If you want to limit the use of this procedure to database connections that have a database connection ID of `manager` or `salesrep`, you can modify the procedure shown in `i-adcus2.p` to authorize access based on the database connection ID for single connected database:

i-adcus2.p



The first part of `i-adcus2.p` authorizes the user to run the procedure. The `CAN-DO` function compares the values listed in the parentheses against the database connection ID for a single connected database. If the database connection ID does not match any of the values listed, the procedure displays a message and exits. If the database connection ID does match one of the values (listed user IDs), the procedure continues executing.

The user ID pattern list you provide in the `CAN-DO` function is a comma-separated list of user ID tokens. You can use tokens to indicate specific users who have or do not have access. The following table lists the types of tokens you can specify.

Table 18: Values to use for user ID pattern lists

Value	Meaning
`*`	All users are allowed access
`user`	This user has access
`!`*`user`*	This user does not have access
`string``*`	Users whose IDs begin with *`string`* have access
`!`*`string`*`*`	Users whose IDs begin with *`string`* do not have access

You can use any combination of patterns to define user ID pattern lists. These patterns are fully consistent with the permissions that you can define for OpenEdge database tables and fields using database administration tools.

The following table shows some examples of fully qualified user ID matches.

Table 19: Fully-qualified user ID matches

This pattern...	Matches...
	The blank user name in the blank domain
`*`	All user IDs
`mark@acme.com`	The `"mark"` user name in the `"acme.com"` domain
`mark`	The `"mark"` user name in the blank domain

This pattern...	Matches...
`mark*`	All user names that begin with `"mark"` in the blank domain
`*jones`	All user names that end with `"jones"` in the blank domain
`db*user`	All user names that begin with `"db"` and end with `"user"` in the blank domain
`@acme`	The blank user name in the `"acme"` domain
`*@acme`	All user names in the `"acme"` domain
`@`	The blank user name in the blank domain
`mark@*`	The `"mark"` user name in any domain
`*@*`	Any user name in any domain
`mark*@acme.*`	Any user name that begins with `"mark"` in any domain that begins with `"acme."`
`*@*.admins`	Any user name in any domain that ends with `".admins"`

For a more complete description of the possible patterns and their meanings, see *Introduction to Identity Management* and the Database Administration online help. For more information on the `CAN-DO` function, see *ABL Reference*.

The `USERID` function (with a specified or single database connection) allows you to check user IDs in a procedure. Use the function in `i-adcus3.p` when you want to allow **only one** user ID access to a procedure.

i-adcus3.p

```
IF USERID <> "manager"
THEN DO:                                          } Security
  MESSAGE "You are not authorized to run this procedure.".  } checking
  RETURN.
END.

REPEAT:                                           } p-
  INSERT customer WITH 2 COLUMNS.                 } adcust.p
END.
```

If the user ID of the user running the procedure is not `manager`, the procedure displays a message and exits. If the user ID is `manager`, the procedure continues.

If you use either the `CAN-DO` function or the `USERID` function to compare the specified database connection ID with one or more user IDs you hard-code in a procedure, you must modify and recompile that procedure whenever you change the user IDs allowed access to it. You can avoid having to make these changes by building a permissions table for activities in your application. For more information, see Defining activities-based user authorization on page 159.

Also, given a single string of user IDs, the `CAN-DO` function, by default, validates the single `USERID` (database connection ID) value for a single database connection. If you connect to more than one database, a `USERID` function requires a specified logical database name and a `CAN-DO` function requires a second argument consisting of an explicit `USERID` function with a specified logical database name.

The second argument of the CAN-DO function can also consist of a string expression that specifies an authenticated user ID other than the database connection ID. For example, you can specify the USER-ID attribute of a client-principal object validated against the application trusted domain registry to specify the OpenEdge session ID. In this way, you can use the CAN-DO function to authorize procedure access independent of any database connection. For more information, see Elements for authenticating ABL user identity on page 101. You can also use the list of user roles set for the ROLES attribute on the client-principal object to identify user permissions for the CAN-DO function, for example:

```
IF CAN-DO(hCP:ROLES, "Admin") THEN ...
```

Defining activities-based user authorization

Applications that you write probably break down into several areas or activities. For example, you may have one set of procedures that handles Customer activities, and another set that handles inventory activities. For each set of activities, you may want to establish a valid group of users. To do this, you build a *permissions table*. To establish and use a permissions table, you must:

- Create a table that defines the kinds of access different users have to application activities.
- Include statements in application procedures to check the permissions table at run time.
- Write a procedure that can modify access permissions.

Creating an application activity permissions table

You create a permissions table within the Data Dictionary. These topics call the table permissions, but you can use any name you want. Each record in the permissions table contains at least two fields: an `activity` field and a `can–run` field. The `activity` field contains the name of the procedure and the `can–run` field contains a comma-separated list of the user IDs of those users who have permission to run the procedure. (Whether these are database connection or OpenEdge session IDs depends entirely on how you match them with the CAN-DO function.) Normally, the primary index is built on the activity field.

After you create a permissions table, you must add a record for every application activity for which you want to provide security.

Adding records to the permissions table

To create records for the permissions table, you must first determine the users and the activities. The users are identified by their user IDs, and the activities by specific procedures or subsystems.

In the following example, three user IDs—`manager`, `salesrep`, and `inventory`—are given permission to perform the following activities:

- Add new Customers to the database by running `i-adcust.p`, which `manager` and `salesrep` have permission to run.
- Update records in the database by running `i-chcust.p`, which `manager` and `salesrep` have permission to run.
- Remove Customer records from the database by running `i-delcus.p`, which `manager` has permission to run.
- The order report and mailing label procedures (`i-itlist.p` and `i-rept6.p`) are grouped into a subsystem called `print`, which `manager` and `inventory` have permission to run.

To add these application activities as records to the permissions table, you can write this simple ABL procedure:

i-prmsn.p

```
REPEAT:
  INSERT permissons.
END.
```

This procedure lets you add records of activities to the permissions table until you press **END–ERROR**. The following figure shows records of application activities that you can add to the permissions table.

Figure 10: Sample activity permissions entries

Activity	Can-Run
p-adcust.p	manager, salesrep
p-chcust.p	manager, salesrep
p-delcus.p	manager
print	manager, inventory

After you create these records of application activities, you must include statements in your procedures that check them at run time. After that, the security administrator is responsible for maintaining these records.

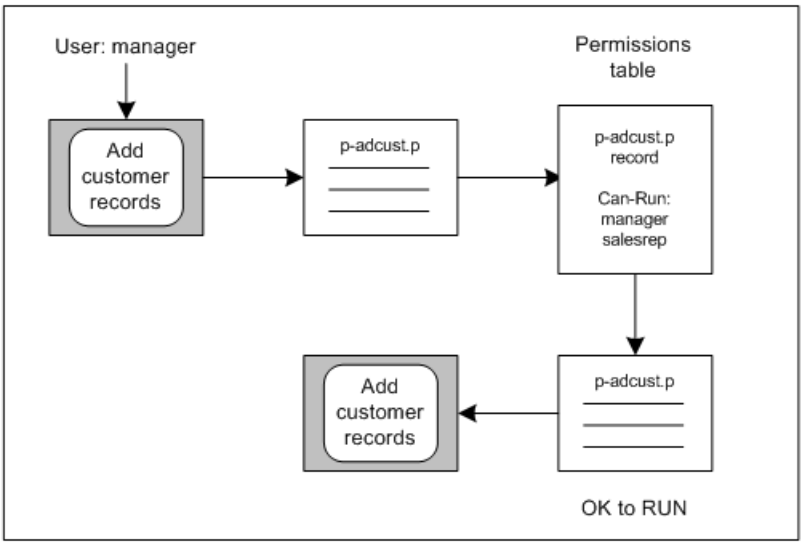
Including security checking in procedures

When the user runs a procedure, you can check the permission for the activity associated with the procedure. Specifically, you:

- Find the activity record in the permissions table.
- Compare the permissions for the activity with the user ID (database connection ID or OpenEdge session ID) of the user running the procedure.
- If there is a match for the user ID, allow access. Otherwise, display a message and exit from the procedure.

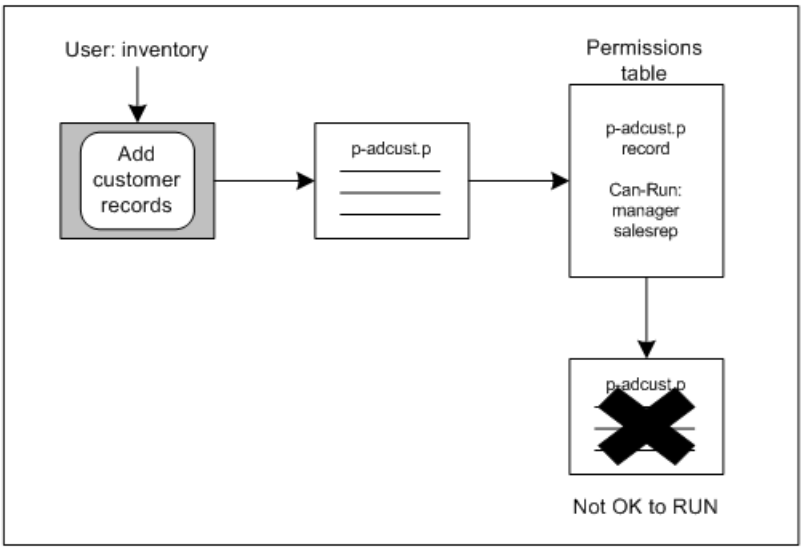
The following figure shows what happens when the user with user ID `manager` runs the `i-adcust.p` procedure.

Figure 11: User with permission to run a procedure



When the specified user ID and the permission defined in the `i-adcust.p` record in the permissions table match, the that user can run the procedure. The following figure shows what happens when the user with user ID `inventory` tries to runs `i-adcust.p`. Because there is no match, the procedure displays a message and the user cannot run the remaining code.

Figure 12: User without permission to run a procedure



You use the `CAN-DO` function to do security checking in your procedure. The procedure `i-adcus4.p` is a modified version of the `i-adcust.p` procedure that includes activity-based security checking.

i-adcus4.p

```
/* Security checking */
DO FOR permission:
  FIND permission "p-adcust.p" NO-LOCK.
  IF NOT CAN-DO(can-run) THEN DO:
    MESSAGE "You are not authorized to run this procedure.".
    RETURN.
  END.
END.

/* p-adcust.p */
REPEAT:
  INSERT Customer WITH 2 COLUMNS.
END.
```

The first part of this procedure makes sure the user's single database connection ID is authorized to run the procedure. The `FIND` statement reads the permission record for the `i-adcust.p` procedure. The `CAN-DO` function compares the value of the `can-run` field in the record with the database connection ID of the user running the procedure. If the values do not match, the procedure displays a message and exits. If there is a match, the procedure allows the user to add Customer records.

The AVM checks privileges within a `DO FOR` block to ensure that the record read by the `FIND` statement is held only during that block, rather than during the entire procedure. In addition, the `NO-LOCK` option ensures that other users can access or update the permissions table while this procedure is running.

The part of the `i-adcust.p` procedure that does security checking is standard. For example, you could include the same security checking statements in the procedures `i-chcust.p` and `i-delcus.p`, if you change the name of the activity record being read in the permissions table.

i-delcs2.p

```
/* Security checking */
DO FOR permission:
  FIND permission "p-delcs.p" NO-LOCK.
  IF NOT CAN-DO(can-run) THEN DO:
    MESSAGE "You are not authorized to run this procedure.".
    RETURN.
  END.
END.

/* p-delcs.p */
PROMPT-FOR Customer.CustNum.
FIND Customer USING Customer.CustNum.
DELETE Customer.
```

The procedure in `i-itlst2.p` shows how you can modify a print procedure, such as `i-itlist.p`, and add security checking to it.

i-itlst2.p

```
/* Security checking */
DO FOR permission:
  FIND permission "print" NO-LOCK.
  IF NOT CAN-DO(can-run) THEN DO:
    MESSAGE "You are not authorized to run this procedure.".
    RETURN.
  END.
END.

/* p-itlst2.p */
FOR EACH Customer NO-LOCK WHERE Customer.CreditLimit > 50000
  BY Customer.PostalCode:
  DISPLAY Customer.Name Customer.Address Customer.City Customer.State
    Customer.PostalCode Customer.CreditLimit
    WITH SIDE-LABELS.
  FOR EACH Order OF Customer NO-LOCK:
    DISPLAY Order WITH SIDE-LABELS.
    FOR EACH OrderLine OF Order NO-LOCK, Item OF OrderLine NO-LOCK:
      DISPLAY OrderLine.LineNum Item.ItemNum Item.ItemName
        OrderLine.Qty OrderLine.Price WITH SIDE-LABELS.
    END.
  END.
END.
```

Here, the `FIND` statement reads the print record from the permissions table. The `CAN-DO` function compares the value of the `can-run` field with the database connection ID of the user running the procedure. If there is no match, the procedure displays a message and exits. If there is a match, the procedure displays order information.

Remember, there is no separate record in the permissions table for the `i-itlst2.p` procedure. However, you can use the record for the `"print"` activity to handle security for any procedure that you specify as a print activity. You can include the same security checking statements in any other procedure that you consider to be a print activity, such as `i-rept6.p`.

i-rept6.p

```
OUTPUT TO mail.lst.
FOR EACH Customer NO-LOCK
  WHERE Customer.Balance >= 1400 BY Customer.PostalCode:
  PUT Customer.Contact SKIP
      Customer.Name SKIP
      Customer.Address SKIP.
  IF Customer.Address2 NE "" THEN
    PUT Customer.Address2 SKIP.
  PUT Customer.City + ", " + Customer.State + "  " +
    STRING(Customer.PostalCode," 99999") FORMAT "X(23)" SKIP(1).
  IF Customer.Address2 EQ "" THEN PUT SKIP(1).
END.
```

For application maintenance purposes, you might want to put security checking statements into an include file. Procedures that require security checking can simply include that file, passing the activity as an argument. An example of a such an include file is `i-chkprm.i`.

i-chkprm.i

```
DO for permissions:
  FIND permissions {1} NO-LOCK.
  IF NOT CAN-DO (can-run) THEN DO:
    MESSAGE "You are not authorized to run this procedure".
    RETURN.
  END.
END.
```

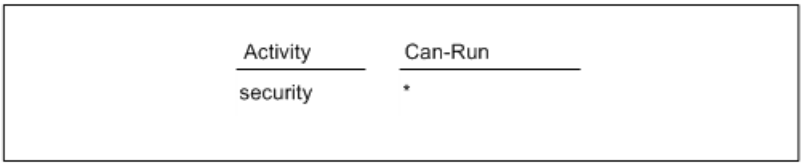
Protecting and maintaining the permissions table

To protect the permissions established in the permissions table, you can provide the security administrator with the following procedures:

- A procedure that defines who can modify the permissions table. Initially, you can run this procedure and enter the user ID of the security administrator, as well as database connection IDs of those authorized to modify the permissions table.
- A security update procedure (for example, `i-secupd.p`) that the security administrator or other authorized users can run to modify permissions for specific procedures and functions.

To do security checking, these procedures require a record in the permissions table associated with the activity of maintaining security. The following figure shows an example security record in the permissions table.

Figure 13: Sample security record for activity permissions



When you create the security record, initialize the `can-run` field with an asterisk. This means that, initially, any user can run the security administration procedure (`i-secadm.p`). However, after you run it and enter the authorized user IDs, only the authorized users can change the security record.

i-secadm.p

```
/* Security checking */
FIND permission "security".
IF NOT CAN-DO(can-run) THEN DO:
  MESSAGE "You are not authorized to run this procedure.".
  RETURN.
END.

/* i-secadm.p */
DISPLAY "Please enter one or more security ids for the security
 administrator, separating the ids with commas" SKIP(1).
UPDATE can-run.
```

The authorized users can also run the next procedure, `i-secupd.p`, which updates permissions for procedures and activities.

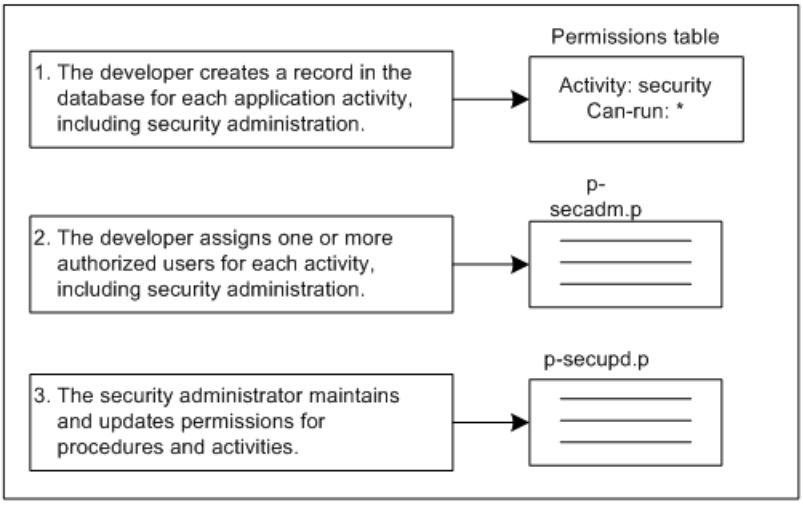
i-secupd.p

```
/* Security checking */
DO FOR permission:
  FIND permission "security" NO-LOCK.
  IF NOT CAN-DO(can-run) THEN DO:
    MESSAGE "You are not authorized to run this procedure.".
    RETURN.
  END.
END.

/* i-secupd.p */
REPEAT FOR permission:
  PROMPT-FOR activity.
  FIND permission USING activity.
  UPDATE can-run.
END.
```

The first part of `i-secupd.p` checks the security record in the permissions table to make sure that the user is authorized to run the procedure. If the user is not authorized, the procedure displays a message and exits. Otherwise, the second part of `i-secupd.p` permits the user to modify the `can-run` field for a specified activity.

The following figure summarizes the security process developed for procedures and functions in an application.

Figure 14: Summary of run-time authorization process



Authorizing user access to tables and fields

OpenEdge can authorize user access to tables and fields of a given OpenEdge RDBMS according to permissions settings available through the OpenEdge Data Administration or character-mode Data Dictionary tools. These settings can authorize specific users to read, write, create, delete, dump, and load table records, or to read and write individual table fields. These settings can apply at both compile time and run time, or at compile-time only, at your option. For more information on configuring these settings and using them for compile-time authorization, see the sections on security in *Manage ABL Applications*.

Note: Permissions settings for the `_Sequence` metaschema table apply both during compile-time, and during run-time.

When you choose to make these settings apply at run time (using Data Administration), OpenEdge automatically enforces the configured permissions during procedure execution for the current connection ID of a given connected database. Any attempt by the application to affect tables and fields contrary to the authorized settings returns an error to the application.

To avoid returning these data access errors within a procedure, you can test a given dynamic record buffer or field to determine if a specific permission is granted to the current connection ID for the database table associated with the specified dynamic record buffer or field.

To test the run-time permissions of a database record buffer or field, return the logical value of the appropriate `CAN-*` attribute on the corresponding buffer object handle or buffer-field object handle. If the permission is allowed to the user running with the current database connection ID, the attribute returns `TRUE`.

Note: These attributes do not apply to buffers associated with individual temp-tables.

The following table lists the `CAN-*` attributes that you can use to test these data permissions.

Table 20: CAN-* attributes for testing run-time data access permissions

This attribute...	Indicates if the user can...
`CAN-CREATE`	Create records in the database table associated with the given buffer object
`CAN-DELETE`	Delete records in the database table associated with the given buffer object
`CAN-READ`	Read records or fields in the database table associated with the given buffer object or buffer-field object
`CAN-WRITE`	Write records or fields in the database table associated with the given buffer object or buffer-field object

For example, to test a dynamic record buffer for permission to read records from the associated table before you attempt to query the table, you might use a code fragment like the following:

```
DEFINE VARIABLE bh AS HANDLE NO-UNDO.
DEFINE VARIABLE qh AS HANDLE NO-UNDO.

CREATE BUFFER bh FOR TABLE "customer".

IF bh:CAN-READ THEN DO:
  CREATE QUERY qh.
  qh:SET-BUFFERS(bh).
  qh:QUERY-PREPARE("FOR EACH Customer").
  qh:QUERY-OPEN.
  qh:GET-FIRST.
  DISPLAY bh:NAME.
END.
```

Using ABL callbacks in authentication systems

ABL authentication callbacks allow you to:

- Extend or modify the built-in authentication systems where OpenEdge performs the user authentication (`_oeusertable`, `_oslocal`)
- Extend an SSO-only authentication system, including the built-in `_extsso` or a user-defined authentication system that is not enabled for OpenEdge-performed user authentication
- Design and implement a user-defined authentication system that is enabled for OpenEdge-performed user authentication

Advantages of using ABL authentication callbacks

ABL authentication callbacks provide flexibility in authentication implementation and configuration. You can distribute multiple authentication systems. Administrators can choose which authentication system to configure when deploying an application.

They also allow you to separate authentication logic from business logic, which results in reusable authentication code. In addition, since the authentication is not embedded in an ABL application, you can avoid updating and redeploying application code.

Running ABL callbacks

An ABL authentication callback executes as an external procedure consisting of three parts: a main block; a procedure you create named `AuthenticateUser`; and an optional procedure you can create named `AfterSetIdentity`.

The main block runs during the OpenEdge-performed user authentication process. For built-in systems, it is called after user-authentication is complete but before the client-principal object is sealed by OpenEdge. For user-defined systems, it is called at the start of OpenEdge user authentication and also before the client-principal object is sealed.

The main block is automatically called as part of the `CONNECT` statement, the `SETUSERID( )` function, the `SET-DB-CLIENT( )` function, or the `SECURITY-POLICY:SET-CLIENT( )` method, The main block is deleted before any of these processes return control to the ABL application.

`AuthenticateUser` is called when OpenEdge-performed user authentication takes place and before the client-principal object is sealed. If the procedure returns success then the authentication succeeded and the client-principal is sealed.

`AfterSetIdentity` is an optional procedure that extends the functionality of SSO.

see Creating an ABL authentication callback procedure on page 171 for information on the functionality and the implementation of `AuthenticateUser` and `AfterSetIdentity`.

Adding an ABL callback to an authentication system

You can associate a callback with an authentication system using either the Data Administration Console or the Data Administration tool. Both allow you to specify the name of an ABL authentication callback that is in the PROPATH. (You can also specify a fully-qualified pathname, but that would limit the portability of the configuration.)

In the Data Administration Console, navigate to the **Security Summary** panel of a connected database and click **Edit authentication systems**.

In the Data Administration tool, choose **Admin** > **Security** > **Domain Maintenance** > **Authentication Systems** from the main menu bar after connecting to a database.

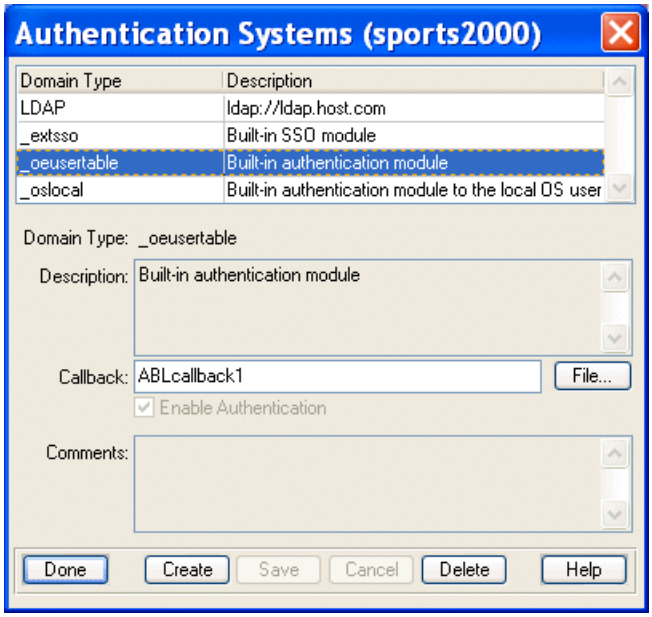
See the online help in each tool for more information.

Extending the built-in systems where OpenEdge performs user authentication

You can extend built-in authentication systems where OpenEdge perform authentication (`_oeusertable` and `_oslocal` for example) by using either the Data Administration Console or the Data Administration tool to configure a built-in authentication system with a callback. (See the online help for each tool for more information.)

The following figure shows the Authentication Systems dialog in the Data Administration tool. It lists the available authentication systems in the **Domain Type** column. Note that, in this case, the built-in `_oeusertable` authentication system is extended by the callback, ABL`callback1` (where ABL`callback1` is a procedure in a folder included in the PROPATH).

Figure 15: _oeusertable configured with an authentication callback procedure



When _oeusertable or _oslocal are configured with an authentication callback, user authentication proceeds as follows:

1. The application launches an OpenEdge-performed user authentication operation.

   For example, it executes the CONNECT statement with specified User ID (-U) and Password (-P) connection parameters. Or it executes the SECURITY-POLICY:SET-CLIENT( ) method with an unsealed client-principal object initialized with the required user credentials.

2. After user credentials are validated, the AVM executes the AuthenticateUser procedure in ABLcallback1. (see Creating an ABL authentication callback procedure on page 171 for more information about AuthenticateUser.)

   If user credentials are not valid, authentication immediately fails and the AVM does not execute the AuthenticateUser procedure. The AVM raises a general authentication failure error and control returns to the application.

   Note that **Enable Authentication** in the above figure is selected by default because _oeusertable (as well as _oslocal) is predefined as enabled for user authentication. This enables the AuthenticateUser procedure to execute and extend the built-in authentication system. This selection cannot be disabled. (Clear the **Callback** field if you do not want callbacks to run.)

3. AuthenticateUser performs whatever further authentication, or other checking that you might require. For example, you could check whether a user has exceeded a set number of logins in a given time period, and generate a customized error message if that condition exists.
4. Since the client-principal object is unsealed at this point, an authentication callback procedure can add additional information (user roles, for example) to the object.
5. If the AuthenticateUser procedure returns success, the AVM seals the client-principal object in the LOGIN state. The AVM uses SSO to assign the authenticated identity to a database connection or an ABL session.
6. After using SSO to assign user identity, the AVM executes any optional actions specified in the AfterSetIdentity procedure in ABLcallback1.

   The actions specified in AfterSetIdentity extend the default SSO authentication process. However, since the client-principal object is sealed, you cannot change identity settings. You can initiate other actions, such as changing context information stored in an OpenEdge database. (See Creating an ABL authentication callback procedure on page 171 for more information about AfterSetIndentity.)

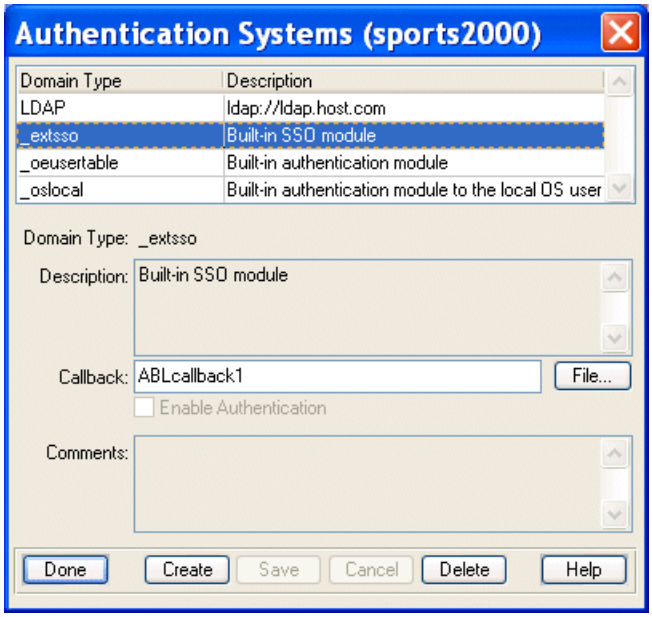
If the application later uses the sealed client-principal from step 5in another successful SSO operation, AfterSetIdentity also executes as in step 6.

Extending SSO-only authentication systems

The primary purpose of SSO-only authentication systems is to enable SSO for application-performed authentication. The built-in _extsso authentication system is an example of an SSO-only authentication system. You can extend _extsso by using either the Data Administration Console or the Data Administration tool to configure the built-in authentication system with a callback. (See the online help for each tool for more information.)

The following figure shows the Authentication Systems dialog in the Data Administration tool. It lists the available authentication systems in the **Domain Type** column. In this case, the built-in `_extsso` authentication system is extended by the callback, `ABLcallback1`.

Figure 16: _extsso configured with an authentication callback procedure



Note that `ABLcallback1` is the same callback configured to extend `_oeusertable` in Extending the built-in systems where OpenEdge performs user authentication on page 167. This callback extends any successful OpenEdge SSO operation using this authentication system by executing `AfterSetIdentity`. **Enable Authentication** in the above figure is disabled by default because SSO does not support authentication. Since authentication is disabled, the `AuthenticateUser` procedure in `ABLcallback1` never executes.

When `_extsso` or an SSO-only user-defined authentication system is configured with an authentication callback, SSO proceeds as follows:

1. An ABL application performs all the authentication, including the sealing of the client-principal object. Since **Enable Authentication** is disabled, the `AuthenticateUser` procedure in `ABLcallback1` does not execute.
2. The application invokes an SSO operation to assign the authenticated user identity with the sealed client-principal, using the `SET-DB-CLIENT` function or `SECURITY-POLICY:SET-CLIENT( )` method. After the identity is set, OpenEdge calls the `AfterSetIdentity` procedure from `ABLcallback1` to extend the default SSO authentication process. (see Creating an ABL authentication callback procedure on page 171 for more information about `AfterSetIndentity`.)

   Because the client-principal object is sealed, you cannot modify or add information to the client-principal object. However, you can initiate other actions, such as changing context information stored in an OpenEdge database.

If the application later uses the sealed client-principal from step 1 in another successful SSO operation, `AfterSetIdentity` also executes as in step 2.

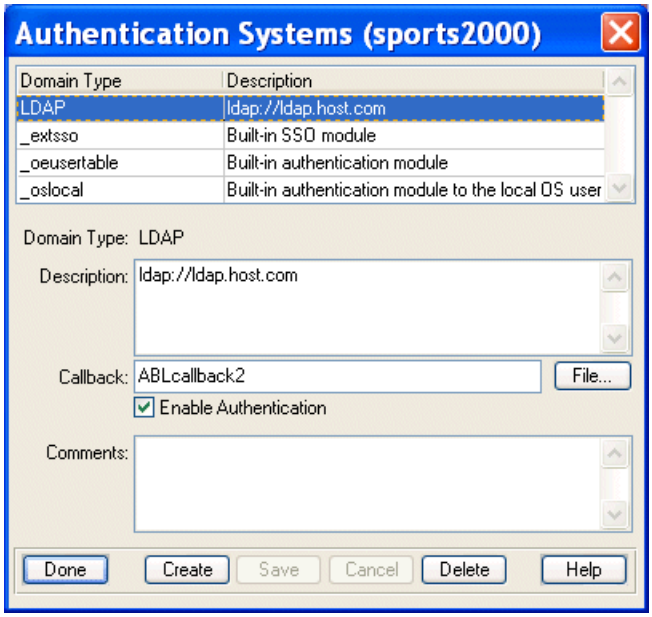
Using ABL to implement user authentication systems

You can implement an application-independent, user-defined authentication system using an ABL callback to do the required user authentication. After programming the callback, you use either the Data Administration Console or the Data Administration tool to configure the authentication system with the callback. (See the online help for each tool for more information.)

The following figure shows the Authentication Systems dialog in the Data Administration tool. It lists the available authentication systems in the **Domain Type** column. In this example, A new, user-defined authentication system, LDAP, is configured to use a callback, `ABLcallback2`, for user authentication. This callback validates user credentials through an external user account system managed by the Lightweight Directory Access Protocol (LDAP).

Note: Although you can use ABL authentication callbacks to configure external authentication services (user verification based on LDAP, for example) in an OpenEdge domain, the details for implementing external authentication services are beyond the scope of this documentation.

Figure 17: A user-defined authentication system implemented by a callback



When LDAP is configured with `ABLcallback2`, user authentication proceeds as follows:

1. The AVM executes the `AuthenticateUser` procedure in `ABLcallback2` in response to an OpenEdge-performed authentication operation, such as the `CONNECT` statement invoked with specified User ID (`-U`) and Password (`-P`) connection parameters or executing the `SECURITY-POLICY:SET-CLIENT( )` method invoked with an unsealed client-principal object initialized with the required user credentials.

2. `AuthenticateUser` processes user credentials by executing a user-validation procedure (based on LDAP in this example).

   Depending on whether the user authentication succeeds or fails, you set an appropriate error code that is returned to the AVM when `AuthenticateUser` execution completes. (see Creating an ABL authentication callback procedure on page 171 for more information about `AuthenticateUser`.)

3. `AuthenticateUser` can perform whatever further authentication, or other checking, that you might require. For example, you might check whether a user has exceeded a set number of logins in a given time period, and generate a customized error message if that condition exists.

   Also because the client-principal is unsealed at this point, the procedure can add additional information (user roles, for example) to the object.

4. If the `AuthenticateUser` procedure returns successfully, the AVM seals the client-principal object in the LOGIN state. The AVM uses SSO to assign the authenticated identity to a database connection or an ABL session.
5. After using SSO to assign user identity, the AVM executes any optional `AfterSetIdentity` actions defined in `ABLcallback2` to extend the default SSO operation.

   Because the client-principal object is sealed, you cannot modify or add information to the client-principal object. However, you can initiate other actions, such as changing context information stored in an OpenEdge database, for example.

If the application later uses the sealed client-principal from step 4 in another successful SSO operation, `AfterSetIdentity` also executes as in step 5.

Creating an ABL authentication callback procedure

An ABL authentication callback procedure contains the following internal procedures:

- **`AuthenticateUser`** — performs user authentication for a user-defined authentication system, or extends the built-in authentication systems where OpenEdge performs user authentication (`_oeusertable` and `_oslocal`.

  User authentication includes locating the user account, verifying and validating the password, loading additional user account information into the client-principal object (roles, for example), verifying if the user account is enabled, checking any account usage restrictions, checking if a new password is required, and so on.

- **`AfterSetIdentity`** — (optional) performs additional functions for an SSO operation invoked by the application, or as part of an OpenEdge-performed user authentication operation. For example, you could add or modify an application's context information with this procedure. However, the client-principal object is sealed when this procedure executes and you cannot make any changes to it.

In addition, note these restrictions to operations on the client-principal object during the execution of authentication callback procedures:

- You cannot delete the client-principal object.
- You cannot call the `AUTHENTICATION-FAILED( )`, `EXPORT-PRINCIPAL( )`, `IMPORT-PRINCIPAL( )`, `INITIALIZE( )`, `LOGOUT( )`, or `SEAL( )` methods.
- You cannot change the `DOMAIN-NAME`, `DOMAIN-TYPE`, `PRIMARY-PASSPHRASE`, `QUALIFIED-USER-ID`, `SESSION-ID`, or `USER-ID` attributes.
- You cannot change the identity of any existing database connection with the `SETUSERID` function.

AuthenticateUser and AfterSetIdentity signatures

In an ABL authentication callback, the `AuthenticateUser` and the `AfterSetIdentity` procedures must be implemented with the following signatures:

```
PROCEDURE AuthenticateUser:
DEFINE  INPUT    PARAMETER  hCP             AS HANDLE.
DEFINE  INPUT    PARAMETER  cSystemOptions  AS CHARACTER EXTENT.
DEFINE  OUTPUT   PARAMETER  iPAMStatus      AS INTEGER INIT ?.
DEFINE  OUTPUT   PARAMETER  cErrorMsg       AS CHARACTER.
END.

PROCEDURE AfterSetIdentity:
DEFINE  INPUT    PARAMETER  hCP             AS HANDLE.
DEFINE  INPUT    PARAMETER  cSystemOptions  AS CHARACTER EXTENT.
END.
```

Note that `AfterSetIdentity` is optional. Include it only if you intend to extend the default SSO operation.

The following is a description of the parameters in the signatures:

- **`hCP`** — A handle to the client-principal object containing the user's identity information. The client-principal object is passed to the `SET-DB-CLIENT( )` function or `SECURITY-POLICY:SET-CLIENT( )` method. OpenEdge creates a client-principal for user authentication in response to a `CONNECT( )` statement that executes with the User ID (`-U`) and Password (`-P`) connection parameters.
- **`cSytemOptions`** — A character array containing any system options defined in the OpenEdge security domain that is pointed to by the client-principal object. System options are set when you configure a domain in the Data Administration tool or in the Data Administration Console.

  Note: Each authentication system implements its own unique set of system options. Refer to the authentication system's documentation to determine which options are supported.

  The value specified in `cSytemOptions` is user-defined but must follow the correct syntax. They must follow the startup parameter syntax, where a parameter is always precede by a dash (-) and each token must be separated by a space character (for example, `-myparam1 -myparam2 value2`). Each element of the array will contain a single parameter followed by an argument (if one was specified). The parameters are defined in `_sec-authentication-domain`, a meta-schema table used by the Data Administration tool and the Data Administration Console to store domain configuration information.

  OpenEdge reserves two options to indicate whether the application is authenticating the user for a session (`-setClient`) or for an OpenEdge database connection (`-db`*`logical-db-name`*). The `-setClient` option is present in the array when the application is executing `SECURITY-POLICY:SET-CLIENT( )` during authentication, and if OpenEdge is setting the credentials for the session during the `AfterSetIdentity` operation. The `-db`*`logical-db-name`* is present in the array when `SET-DB-CLIENT( )` or the `CONNECT` statement, or if OpenEdge is trying to set the credentials for a database connection while executing `SECURITY-POLICY:SET-CLIENT( )`. AfterSetIndentity can use these options to determine whether the operation is for a session or for an OpenEdge database connection.l

  If the value specified in `cSytemOptions` is syntactically incorrect, a configuration error occurs when you try to authenticate the user using that domain.

- **`iPAMStatus`** — An integer status code that you return to the AVM to indicate whether user authentication succeeded or not. This must be one of the values implemented as static properties of the `Progress.Security.PAMStatus`. (See *ABL Reference* for more information.) If a status code other than `Success` is returned from `AuthenticateUser`, the user authentication operation ends with failure, and

the AVM takes no further action except to seal the client-principal object in the FAILED state and return the error to the application.

Note: When extending a built-in user authentication operation for `_osusertable` or `_oslocal`, `AuthenticateUser` runs **only** if the built-in user authentication is successful. Otherwise, the operation ends without running `AuthenticateUser` and returns a general authentication failure error to the application.

- **`cErrorMsg`** — An error message that you create. It is used as the error message generated by `SET-DB-CLIENT( )`or `SECURITY-POLICY:SET-CLIENT( )`, if the returned status (`iPAMStatus`) is `Custom`.

ABL authentication callback procedure example

The `authentication_callback.p` example shows how to set up an ABL authentication callback procedure implemented with the appropriate type-name reference (`Progress.Security.*.`)for the PAMStatus class and with basic error handling. Note that the `/* Perform validation */` comment is a placeholder for the section where you would add your own validation code.

You can use `authentication_callback.p` as a template for your own authentication callback.

authentication_callback.p

```
/* so that errors are thrown back to the main block */
ROUTINE-LEVEL ON ERROR UNDO, THROW.

USING Progress.Security.*.

/* Internal Procedures */
/* Procedure:   AuthenticateUser
 * Description: Perform the user account authentication action using the
 *              information in the Client-Principal object:
 *                 a) Establishes the presence of the user account
 *                 b) Validates the password(s)
 *                 c) Optionally load user account information into the
 *                       Client-Principal object.
 *              Can also perform the user account restriction checking
 *              using the information contained in the Client-Principal object:
 *                 a) account enabled
 *                 b) account day/time restrictions
 *                   c) Client workstation/tty access
 *                   d) Requirement for a new password
 *              NOTE: The ABL runtime will seal the Client-Principal object
 *              after the callback procedure returns.
 *
 */
PROCEDURE AuthenticateUser:

DEFINE INPUT  PARAMETER hCP              AS HANDLE NO-UNDO.
DEFINE INPUT  PARAMETER cSystemOptions   AS CHARACTER EXTENT NO-UNDO.
DEFINE OUTPUT PARAMETER iPAMStatus       AS INTEGER INITIAL ? NO-UNDO.
DEFINE OUTPUT PARAMETER cErrorMsg        AS CHARACTER NO-UNDO.

/* Local variables */

DEFINE VARIABLE iActionStatus AS INTEGER NO-UNDO.
DEFINE VARIABLE checkStop     AS LOGICAL NO-UNDO.

DO ON STOP UNDO, LEAVE
   ON QUIT UNDO, LEAVE :

    iActionStatus = ?.

    checkStop = YES.

/* Perform validation */

    checkStop = NO.

    CATCH e AS Progress.Lang.Error:
        /* handle errors - possibly log error or set cErrorMsg */
       iActionStatus = PAMStatus:Custom.
       cErrorMsg = "Failed to process callback:" + e:GetMessage(1).
    END.
END.
```

```
 IF checkStop THEN DO:
    ASSIGN iActionStatus = PAMStatus:Custom
           cErrorMsg = "STOP Condition found".
 END.
 IF iActionStatus == ? THEN
   iPAMStatus = PAMStatus:Success.
 ELSE
   iPAMStatus = iActionStatus.
 END.
END.

/* Procedure: AfterSetIdentity
 * Description: This is a notification that the AVM has set the credentials
 *              for the session or an OpenEdge database connection.
 */
PROCEDURE AfterSetIdentity:

DEFINE INPUT PARAMETER hCP  AS HANDLE NO-UNDO.
DEFINE INPUT PARAMETER ss   AS CHARACTER EXTENT NO-UNDO.

END.
```

Using cryptography to secure data

Cryptography provides a means to secure data from unauthorized access. Using cryptographic functions, you can perform such operations as "scramble" (*encrypt*) the initially human-readable (*clear text*) contents of a character string so that the contents of the string is no longer humanly readable. If you are authorized, you can later "unscramble" (*decrypt*) such encrypted data so that it is humanly readable again. You can also ensure the integrity of data to verify that it has not been changed in an unauthorized manner.

The science of cryptography offers many techniques for securing data. OpenEdge supports symmetric key encryption and decryption to make data confidential as well as one-way hashing to generate password-based encryption (PBE) keys and generate values for verifying data integrity in ABL.

Using these ABL cryptographic facilities, you can:

- Create and maintain a consistent cryptography policy that determines how cryptography is managed in your application
- Generate cryptographic keys, the "secret" codes used to seed the ciphers (algorithms) used to encrypt and decrypt data and that can become part of your cryptography policy
- Encrypt and decrypt data based on a choice of standard cryptographic algorithms that can also become a part of your cryptography policy
- Transport and manipulate encrypted data in binary or character form
- Create and use message digests to guarantee the integrity of data that you move from one place to another

---

Caution: Do not use the cryptographic facilities described in this section unless you are thoroughly versed in the use of cryptography within a comprehensive application security framework. Incorrect use of these facilities can result in corrupted or otherwise lost and permanently unrecoverable data.

For a general overview of cryptography and how you can use cryptography in OpenEdge, see *Introduction to Security and Auditing*.

These sections describe the basic ABL cryptographic facilities available and how you can use them in an ABL application.

- Creating and maintaining a cryptography policy on page 176
- Generating encryption keys on page 178
- Implementing symmetric cryptography in ABL on page 181
- Using message digests in ABL on page 185
- Managing and transporting crypto data on page 186

Creating and maintaining a cryptography policy

ABL allows you to establish a consistent set of defaults for handling cryptography in an application using the SECURITY-POLICY system handle. These cryptography settings apply globally for the duration of an ABL session. The following table lists the attributes that you can read and set for this system handle. These attributes are both readable and setable unless otherwise indicated.

Table 21: SECURITY-POLICY system handle attributes

Attribute	Description
ENCRYPTION-SALT	A RAW value used together with a simple password to generate a more random and unique password-based encryption (PBE) key (default: none). For any value that you set, only the first 8 bytes are used, and values with fewer than 8 bytes are padded with zeroes. **Note:** Typically, you never need to set this value, and then only if the security specification for your application requires it.
PBE-HASH-ALGORITHM	A CHARACTER value that specifies the hashing algorithm to use for generating a PBE key, which you can specify as "MD5", "SHA-1", "SHA-256", or "SHA-512" (default: "SHA-1"). **Note:** Typically, you never need to set this value, and then only if the security specification for your application requires it.
PBE-KEY-ROUNDS	A positive INTEGER value that specifies the number of algorithm iterations to use for PBE key generation (default: 1000). **Note:** Extremely large values for this setting can significantly degrade performance. Typically, you never need to set this value, and then only if the security specification for your application requires it.

Attribute	Description
SYMMETRIC-ENCRYPTION-ALGORITHM	A CHARACTER value that specifies a supported algorithm, mode, and key size used by the ABL symmetric encryption and decryption facilities. It must be one of the comma-separated values from the SYMMETRIC-SUPPORT attribute (default: "AES_CBC_128").
SYMMETRIC-ENCRYPTION-KEY	A write-only RAW value that specifies the symmetric key to use for encryption and decryption (default: none).
SYMMETRIC-ENCRYPTION-IV	A RAW value used together with the symmetric encryption key to provide a more random data encryption (default: none). **Note:** Typically, you never need to set this value, and then only if the security specification for your application requires it.
SYMMETRIC-SUPPORT	A read-only CHARACTER value that contains a comma-separated list of supported values for the SYMMETRIC-ENCRYPTION-ALGORITHM attribute setting.

For information on OpenEdge-supported defaults and options for setting these attributes, see *ABL Reference*.

Caution: In general, you must manage the values that you set for a cryptography policy in a safe manner, including generation, storage, and transport. Improper management of these values can result in loss of data.

Setting a symmetric encryption algorithm

The encryption algorithm setting determines how the ABL encryption and decryption functions (see Implementing symmetric cryptography in ABL on page 181) work to protect data confidentiality in your application. This setting is actually a combination of three elements expressed in the following form:

Syntax

```
AAA_MMM_n
```

These three elements specify:

AAA

An alpha or alphanumeric abbreviation for the algorithm.

MMM

An alpha abbreviation for the mode to use.

n

A numeric value for the key size to use.

The OpenEdge default setting for the SYMMETRIC-ENCRYPTION-ALGORITHM attribute is "AES_CBC_128", which means the AES algorithm in CBC mode using a 128-bit key. The combination of algorithm, mode, and key size determines the strength and speed of the encryption. You can accept the OpenEdge default or choose a different setting, depending on your application requirements. Any setting you choose for the SYMMETRIC-ENCRYPTION-ALGORITHM attribute is used by each instance of an ABL encryption/decryption function unless you specify an override for that function.

For more information on encryption algorithms, modes, and cryptographic keys, see the sections on cryptography in *Introduction to Security and Auditing*.

Setting an encryption key

You can set the SYMMETRIC-ENCRYPTION-KEY attribute to any RAW, MEMPTR, CHARACTER, or LONGCHAR value that conforms to the specified key size of your algorithm setting. A RAW or MEMPTR value specifies a raw binary value for the key. A CHARACTER or LONGCHAR value creates a PBE, PKC#5-compliant, binary key with a default salt value.

Caution: Note that the result of setting the SYMMETRIC-ENCRYPTION-KEY attribute is unreadable to prevent it against access by memory scanners and debuggers. You must ensure that any source for this value is overwritten in memory, or otherwise protected from unauthorized access.

ABL also provides a number of facilities for generating symmetric keys of various types. For more information, see Generating encryption keys on page 178.

Generating encryption keys

ABL provides a set of built-in key generation functions for quickly and effectively generating keys for symmetric encryption of the right size and content for a given algorithm. You can also use these functions to generate keys for any other encryption task, such as for use in generating message digests (see Using message digests in ABL on page 185). Using these functions you can generate two basic types of keys:

- Random keys
- Password-based encryption (PBE) keys

The strength of a cryptographic key depends on its randomness. ABL supports these key generation functions using specially tailored random number generators. For symmetric encryption itself as well as for the generation of PBE keys used in encryption, you can increase the effective key randomness by using an additional key value for both encryption and PBE key generation:

- For symmetric encryption, ABL provides the option of combining an *initialization vector* (SECURITY-POLICY:SYMMETRIC-ENCRYPTION-IV attribute) with the symmetric key (SECURITY-POLICY:SYMMETRIC-ENCRYPTION-KEY attribute) to perform data encryption. By combining it with the symmetric key, an *initialization vector* increases effective key randomness, and therefore increases the strength of the encryption.
- For generating PBE keys used in symmetric encryption, ABL provides the option of combining the user-supplied password with a *salt* that you can set as the value of the SECURITY-POLICY:ENCRYPTION-SALT attribute. A *salt* is an especially useful random value for generating PBE keys because passwords are typically very simple and often duplicated. A random salt ensures that the PBE key generated for a given password is always unique for each use of the same password.

Thus, the ABL key generation functions allow you to generate symmetric encryption results whose randomness can be compounded and tailored for the algorithms that you have initialized for the SECURITY-POLICY system handle, as shown in the following table.

Table 22: Cryptographic key generation functions

ABL function	Description
GENERATE-PBE-KEY(*password* [, *salt*])	Evaluates to a RAW password-based key value with the number of bytes determined by the setting of the SECURITY-POLICY:SYMMETRIC-ENCRYPTION-ALGORITHM attribute. This function uses the PKCS#5/RFC 2898 standard for generating a symmetric encryption key based on the one-way hashing algorithm specified by SECURITY-POLICY:PBE-HASH-ALGORITHM. You can use this function to set one or both of these SECURITY-POLICY handle attributes with a value based on a password: • SYMMETRIC-ENCRYPTION-KEY • SYMMETRIC-ENCRYPTION-IV You must specify a character string-based value other than the Unknown value (?) for *password*, and you must maintain the same code page in order to use this function to recover the same key value using this *password*. Specify an 8-byte RAW value for *salt* in order to help ensure that the PBE key generated using *password* is unique for all other uses of the same *password* value. To obtain a random value for *salt* that is most likely to yield a unique PBE key, you can use the GENERATE-PBE-SALT function to generate the value. If you do not specify the *salt* option, the function uses any salt value (other than the Unknown value (?)) that you have set for the ENCRYPTION-SALT attribute of the SECURITY-POLICY handle. **Note:** You can use this function to set the SYMMETRIC-ENCRYPTION-KEY attribute directly as long as the same SYMMETRIC-ENCRYPTION-ALGORITHM setting, PBE-HASH-ALGORITHM setting, *password*, salt value, and code page are used to generate the PBE key value for both encryption and decryption.
GENERATE-PBE-SALT	Evaluates to a RAW random 8-byte value according to the setting of the SECURITY-POLICY:PBE-HASH-ALGORITHM attribute. You can use this function to set one or both of these values: • The ENCRYPTION-SALT attribute of the SECURITY-POLICY handle. • The *salt* parameter of the GENERATE-PBE-KEY function. **Note:** Be careful when using this function to directly set the *salt* parameter of the GENERATE-PBE-KEY function. If you do not save the generated PBE key value separately, but use it directly for encryption, this results in a nonrecoverable key value, rendering the encrypted data unusable. For the PBE key to be recoverable, you must be able to recover both the salt and the password that you combined to initially generate the key value. For more information, see Implementing symmetric cryptography in ABL on page 181.
GENERATE-RANDOM-KEY	

ABL function	Description
	Evaluates to a RAW random key value calculated by a pseudo random number generator (PRNG) and containing the number of bytes determined by the setting of the SECURITY-POLICY:SYMMETRIC-ENCRYPTION-ALGORITHM attribute. You can use this function to set one or both of these SECURITY-POLICY handle attributes with a random value: • SYMMETRIC-ENCRYPTION-KEY • SYMMETRIC-ENCRYPTION-IV **Note:** Do not use this function to directly set the SYMMETRIC-ENCRYPTION-KEY attribute. Because this attribute is not readable, setting it directly from this function leads to a nonrecoverable key value, rendering the encrypted data unusable.

Caution: You must ensure that all values and system conditions required to generate a symmetric key value are available to generate the cryptographic keys used for both encryption and decryption of the same data. Otherwise, encrypted data can become undecipherable and effectively lost.

For more information on symmetric keys and their generation, see *Introduction to Security and Auditing*.

Note: ABL also supports the encryption of keys especially for use in auditing, for example, to secure message authentication code (MAC) keys used to seal audit data for audit archiving. For more information on MACs, see Using message digests in ABL on page 185. For more information on encrypting MAC keys for secure audit archiving, see Custom audit archiving tools on page 257.

Implementing symmetric cryptography in ABL

Once you have established an appropriate cryptography policy for your application, including the necessary key values and algorithms, you can begin to apply encryption and decryption to your data. However, you must also thoroughly plan the role that cryptography plays in your application in order to use it safely and effectively. While ABL provides powerful tools for generating keys and encrypting data, you must manage these tools carefully in your application to avoid serous problems that could result in loss of data.

Caution: Before encrypting any data that you intend to have decrypted, be certain that you can recover the unencrypted data if either the encryption or decryption fails.

For more information, in general, on managing data encryption and decryption, see the *Security* whitepaper prepared by OpenEdge Marketing, which can be found on the PSDN Web site.

Using ABL encryption and decryption functions

ABL provides two built-in functions, DECRYPT and ENCRYPT, to encrypt and decrypt data. Both functions rely on cryptographic parameters that you set using the SECURITY-POLICY system handle or using options for invoking each function.

These functions have the following syntax:

Syntax

```
ENCRYPT( data-to-encrypt [ , encrypt-key [ , iv-value [ , algorithm ]]] )
```

```
DECRYPT( data-to-decrypt [ , encrypt-key [ , iv-value [ , algorithm ]]] )
```

You can provide the *data-to-encrypt* as a CHARACTER, LONGCHAR, RAW, or MEMPTR variable, and the ENCRYPT function evaluates to a MEMPTR value containing the encrypted binary byte stream of the data. You can provide the *data-to-decrypt* as a MEMPTR or RAW variable containing an encrypted binary byte stream, and the DECRYPT function evaluates to a MEMPTR value containing the decrypted binary byte stream of the data. You can then convert the decrypted MEMPTR value to a CHARACTER or LONGCHAR in order to make the decrypted data human-readable. If you need to encrypt and decrypt more than one value, you can marshall these items to and unmarshall them from a MEMPTR using PUT-*datatype* statements and GET-*datatype* functions.

If you specify a parameter value for any of the function options, it overrides any setting of the corresponding SECURITY-POLICY handle attribute. In order to successful decrypt an encrypted value (resulting in a DECRYPT return value identical to the data-to-encrypt value that you input to ENCRYPT), you must invoke both functions using identical cryptographic parameters as shown in the following table, whether you supply them as function options or as attributes of your application cryptography policy.

Table 23: ENCRYPT/DECRYPT function parameters

Set this cryptographic parameter using...	This function option...	Or this SECURITY-POLICY handle attribute...
Symmetric key	encrypt-key	SYMMETRIC-ENCRYPTION-KEY
Initialization vector	iv-value	SYMMETRIC-ENCRYPTION-IV
Algorithm, mode, and key size	algorithm	SYMMETRIC-ENCRYPTION-ALGORITHM

For many applications, you do not need to specify an initialization vector and can use the default algorithm, mode, and key size ("AES_CBC_128"). Typically, you only need to set these values to satisfy specific application requirements.

You must handle all generation, storage, transportation, and provision of the symmetric key and all other values required to decrypt data that you have encrypted. This includes maintaining binary keys in the correct byte endian order for the platform or platforms where cryptographic operations occur.

For information on:

- The SECURITY-POLICY system handle, see Creating and maintaining a cryptography policy on page 176
- Key generation, see Generating encryption keys on page 178
- Managing and transporting keys and data, see Managing and transporting crypto data on page 186

Example code using the ENCRYPT and DECRYPT functions

This section shows basic uses of the ABL symmetric cryptography functions and related ABL elements.

The following example shows the use of a random binary key to encrypt the text "Bathtub Pancake Ladybug".

Example 1: Encryption using a random binary key

```
DEFINE VARIABLE binary-key   AS RAW       NO-UNDO.
DEFINE VARIABLE clear-text   AS CHARACTER NO-UNDO
  INITIAL "Bathtub Pancake Ladybug".
DEFINE VARIABLE crypto-value AS RAW       NO-UNDO.

binary-key = GENERATE-RANDOM-KEY.
SECURITY-POLICY:SYMMETRIC-ENCRYPTION-KEY = binary-key.
SECURITY-POLICY:SYMMETRIC-ENCRYPTION-IV = ?.

crypto-value = Encrypt (clear-text).
```

For the receiver of the crypto-value to be able to successfully decrypt the value and recover the clear text requires the following information:

1. The name of the algorithm (the default, AES_CBC_128, used and obtained by reading SECURITY-POLICY:SYMMETRIC-ENCRYPTION-ALGORITHM)
2. The initialization vector (none used, but available by reading SECURITY-POLICY:SYMMETRIC-ENCRYPTION-IV)
3. The binary key value

Note: Setting SECURITY-POLICY:SYMMETRIC-ENCRYPTION-KEY directly from the GENERATE-RANDOM-KEY function leads to an irrecoverable key because this attribute is write-only and GENERATE-RANDOM-KEY returns a different value if called again.

The following example shows the use of a binary key to decrypt the text from the previous example.

Example 2: Decryption using a random binary key

```
DEFINE VARIABLE binary-key   AS RAW       NO-UNDO.
DEFINE VARIABLE clear-text   AS CHARACTER NO-UNDO.
DEFINE VARIABLE crypto-value AS RAW       NO-UNDO.
/* binary-key is set to the same value as in the previous example */
SECURITY-POLICY:SYMMETRIC-ENCRYPTION-KEY = binary-key.
SECURITY-POLICY:SYMMETRIC-ENCRYPTION-IV = ?.
/* crypto-value is obtained by some means */
clear-text = GET-STRING(DECRYPT(crypto-value), 1).
```

The following example shows the use of a password-based key to encrypt the text, "Matter Property Mass Solid". The password is "Migratory Blueberries".

Example 3: Encryption using a password-based key

```
DEFINE OUTPUT PARAMETER salt-value AS RAW NO-UNDO.
DEFINE VARIABLE password     AS CHARACTER NO-UNDO
  INITIAL "Migratory Blueberries".
DEFINE VARIABLE clear-text   AS CHARACTER NO-UNDO
  INITIAL "Matter Property Mass Solid".
DEFINE VARIABLE crypto-value AS RAW       NO-UNDO.
SECURITY-POLICY:SYMMETRIC-ENCRYPTION-ALGORITHM = "AES_CBC_256".
SECURITY-POLICY:PBE-KEY-ROUNDS                 = 2000.
SECURITY-POLICY:PBE-HASH-ALGORITHM             = "SHA-256".
SECURITY-POLICY:ENCRYPTION-SALT                = GENERATE-PBE-SALT.
/* The salt value is passed as an OUTPUT parameter for use by the
 * following password decryption example */
salt-value = SECURITY-POLICY:ENCRYPTION-SALT.
SECURITY-POLICY:SYMMETRIC-ENCRYPTION-KEY = GENERATE-PBE-KEY (password).
SECURITY-POLICY:SYMMETRIC-ENCRYPTION-IV  = ?.
crypto-value = ENCRYPT (clear-text).
```

For the receiver of the crypto-value to be able to successfully decrypt the value and recover the clear text, the following information is necessary:

1. The name of the algorithm (a non-default value specified and obtained by reading `SECURITY-POLICY:SYMMETRIC-ENCRYPTION-ALGORITHM`)
2. The initialization vector (none is used, but can be set using `SECURITY-POLICY:SYMMETRIC-ENCRYPTION-IV`)
3. The number of hashing algorithm iterations to generate the key (a non-default value specified and obtained by reading `SECURITY-POLICY:PBE-KEY-ROUNDS`)
4. The salt value (obtained by reading `SECURITY-POLICY:ENCRYPTION-SALT`)

   Note: Calling `GENERATE-PBE-SALT` function again produces a different salt value and is not useful in generating the binary key necessary to decrypt the text.

5. The hash algorithm used to transform the password into the binary key (a non-default value specified and obtained by reading `SECURITY-POLICY:PBE-HASH-ALGORITHM`)
6. The password text

In place of items mentioned in points 3 through 6, the binary key value itself can be supplied. Because `SECURITY-POLICY:SYMMETRIC-ENCRYPTION-KEY` is write-only, an intermediate `RAW` variable can be used to hold the value returned from the `GENERATE-PBE-KEY` function, or the function may be called again with the same password value (it will return the same result provided the `SECURITY-POLICY` system handle attributes `SYMMETRIC-ENCRYPTION-ALGORITHM, PBE-KEY-ROUNDS`, `PBE-HASH-ALGORITHM`, and `ENCRYPTION-SALT` all have the same values).

The following example shows the use of a password-based key to decrypt the text from the previous example. The password is `"Migratory Blueberries"`.

Example 4: Decryption using a password-based key

```
/* The salt value from the previous password encryption example is passed in
   as an INPUT parameter */
DEFINE INPUT PARAMETER salt-value AS RAW NO-UNDO.
DEFINE VARIABLE clear-text   AS CHARACTER NO-UNDO.
DEFINE VARIABLE crypto-value AS RAW       NO-UNDO.
/* The password is most likely obtained from the user and not in the code as
   shown here. */
DEFINE VARIABLE password     AS CHARACTER NO-UNDO
  INITIAL "Migratory Blueberries".
SECURITY-POLICY:SYMMETRIC-ENCRYPTION-ALGORITHM = "AES_CBC_256".
SECURITY-POLICY:PBE-KEY-ROUNDS                 = 2000.
SECURITY-POLICY:PBE-HASH-ALGORITHM             = "SHA-256".
SECURITY-POLICY:ENCRYPTION-SALT                = salt-value.
SECURITY-POLICY:SYMMETRIC-ENCRYPTION-KEY       = GENERATE-PBE-KEY(password).
SECURITY-POLICY:SYMMETRIC-ENCRYPTION-IV        = ?.
/* crypto-value is obtained by some means */
clear-text = DECRYPT (crypto-value).
```

Using message digests in ABL

A *message digest* is the result of a one-way hashing operation that produces a unique value for a given unique string of data. Its primary use is to detect any unauthorized changes in data (verify its *data integrity*).

For example, suppose you have a database record with 10 sensitive data fields, and you need to determine if any of the fields have been unexpectedly changed. Before you create or update the database record you can hash the 10 sensitive fields into a single message digest and store the digest under the same record key. Each time you read the database record, you can check its data integrity by hashing the same 10 fields and compare the resulting message digest with the one currently stored for that record. If the two message digests do not match exactly, the data has undergone an unauthorized modification.

Securing a message digest

One property of message digests is that they are predictable, yielding the same result every time they are hashed from the same data. To help protect against someone recomputing the message digest along with an unauthorized change to the data, you can provide a *secret key* for the algorithm to hash a value that is unique to both the data and the key. This value is a type of message authentication code (MAC). You cannot easily recreate a MAC without knowing the secret key.

You can use any value for the key that you choose or use the built-in key generation functions to generate a key for you (see Generating encryption keys on page 178). If you store the secret key securely, like any cryptographic key, this results in a secure message digest.

Generating message digests in ABL

ABL provides the following built-in function to generate message digests:

Syntax

```
MESSAGE-DIGEST( hash-algorithm, data-to-hash [, hash-key ] )
```

The *hash-algorithm* is a character string that specifies which hashing algorithm to use. This can be "SHA-1", "SHA-256", "SHA-512", "HMAC-SHA-1", "HMAC-SHA-256", "HMAC-SHA-384", and "HMAC-SHA-512".

For more detail on the MESSAGE-DIGEST function, see the *ABL Reference*.

Managing message digests

If you want to hash multiple data sources (such as several database fields) into a single message digest, you can marshall the individual data sources into a single MEMPTR variable using the PUT-*datatype* functions before invoking the message digest function. To make this work correctly, you must marshall exactly the same data sources using exactly the same order and data formats to generate comparable digests.

Because message digests are generated as a RAW binary byte stream, you must ensure that the byte endian order is maintained cross different hardware platforms.

The management considerations for message digest keys and results is much the same as for any encryption keys and data, except that you do not have to keep track of the code page of data sources. For more information, see Managing and transporting crypto data on page 186.

Managing and transporting crypto data

Crypto data consists of all the information required to maintain secure data. This includes the source data, typically in clear-text form; the encrypted or hashed data; all passwords, key values, and algorithms; and all platform parameters required to maintain the data, such as code pages for character strings and the byte endian order for binary values.

This section describes some of the basic objectives and requirements for managing crypto data in ABL. For more information on the larger and more complex factors that you need to consider when using cryptography, see the *Security* whitepaper prepared by OpenEdge Marketing, which can be found on the PSDN Web site.

Caution: Managing crypto data incorrectly can result in the loss of your original source data. You must ensure that all interactions with this data are both safe and secure.

Objectives for crypto data management

The main objectives for managing and transporting crypto data is the same for all types of encryption and data:

- To ensure that the required level of security is maintained as data is moved and stored
- To ensure that secured data can always be recovered by those who are authorized to recover it

Requirements for crypto data management

The requirements for managing and transporting crypto data depend upon the:

- **Type of cryptography** — In ABL, this can include symmetric cryptography, message digests, and key generation.
- **Media for storage or transport** — This can include static media, such as databases and external files and dynamic media, such as session memory and network connections.

One general requirement for managing encrypted or digested data is to be certain that all keys once generated, are recoverable. For example, if you use the GENERATE-RANDOM-KEY function to set a symmetric key and you use this function to directly set the SYMMETRIC-ENCRYPTION-KEY attribute of the SECURITY-POLICY system handle, the key value that you set is non-recoverable because the SYMMETRIC-ENCRYPTION-KEY attribute is write-only. If you encrypt data using this setting, you can never decrypt it after the ABL session in which you encrypt the data has ended. Therefore, to correctly use the GENERATE-RANDOM-KEY function for symmetric encryption, you must set and maintain the value of a readable variable using this function and then set the SYMMETRIC-ENCRYPTION-KEY attribute using this variable. Of course, you must also ensure that you immediately save the key value in a secure location and erase the value in memory in order to both secure your current ABL session and safely recover the key value at a later time to decrypt whatever data you have encrypted with it.

Crypto data storage

If you use external files to store crypto keys, passwords, and data make certain that you use secure operating system files to do it.

If you use a database to store crypto data, be sure to secure any keys or passwords that you store in the database separately from the data that you have encrypted.

Caution: Never encrypt a password required to access encrypted data using the same password-based encryption (PBE) key used to encrypt the data. Always manage the password separately from the data that you intend to encrypt with it, and always manage the password separately from any salt that you combine with the password to generate the PBE key.

Crypto data management and transport

When you store or transport crypto data, you can do it in two basic forms:

- **Binary byte stream** — Accessible through an ABL MEMPTR or RAW variable. This the native form in which key values and encrypted or hashed data are generated. The primary requirement for binary data is that you maintain byte-endian order for all hardware platforms where you transport the data. You can help to ensure this by using the GET-BYTE and PUT-BYTE functions to read and write the data instead of GET-LONG and PUT-LONG. You can also convert a binary data string into a string of hexadecimal character pairs using the ABL built-in HEX-ENCODE function, transport it to a different hardware platform, and convert hexadecimal character string back to binary on that platform using the ABL built-in HEX-DECODE function.
- **Base64-encoded character string** — Accessible through an ABL CHARACTER or LONGCHAR variable. You can convert binary data into this form using the ABL built-in BASE64-ENCODE function and convert it back to binary using the ABL built-in BASE64-DECODE function. The primary requirement for Base64-encoded and encrypted data is that you remember to decode it from Base64 before you decrypt the data.

Caution: If encrypted data is not in the correct byte order or format, the DECRYPT function has no way of knowing this and will successfully "decrypt" the data to an invalid value.

If you store encrypted data in a database:

- Do not use it for searches or indexes unless duplicates are allowed, as there is no way to know that two different pieces of data have been encrypted to the same value.
- Plan for changes in the size of database items that you store in both clear text and encrypted form. When using AES and DES ciphers, both encryption and Base64-encoding increase the size of stored data.

Caution: If you encrypt data in a database, other applications that use the database, such as Crystal Reports, might not work as you want.

Planning for changes in size for encrypted data

Because AES, DES, and their variants are block ciphers, the data output from encryption functions using these ciphers typically differs in size from the data that is input.

Note: RC4 is a stream cipher in which the data input to an encryption function is always the same size as the data that is output. Therefore, this section does not apply if you are using RC4 as your encryption cipher.

Data size increases at two points:

- During symmetric encryption, using the ABL `ENCRYPT` function
- During conversion from a binary byte stream to a Base64-encoded string, using the ABL `BASE64-ENCODE` function

The formula for change in data size during symmetric encryption is:

```
ems = (((ums / 8) + 1) * 8)
```

In this formula:

- *ems* is the encrypted maximum size
- *ums* is the unencrypted maximum size

The formula for change in data size when Base64-encoding a binary byte stream is:

```
ems = (((bms / 8) + 1) * 8)
```

In this formula:

- *ems* is the Base64-encoded maximum size
- *bms* is the binary maximum size, which is equal to the encrypted maximum size of the data that has been encrypted using the `DECRYPT` function

This increase in size from binary to Base64-encoding is generally 33%. So, encrypted data in a `RAW` or `MEMPTR` variable becomes 33% larger after you convert to a Base64-encoded `CHARACTER` or `LONGCHAR` variable.

3

Multi-tenant ABL

These topics describe how ABL supports multi-tenancy. For an overview of multi-tenancy, see *Introduction to Database Multi-tenancy*. For existing applications written for non-multi-tenant databases, there is often no difference in the ABL required to run with multi-tenant versions of the same databases. That is, after converting a non-multi-tenant database to a multi-tenant database, most, if not all, of the code in an existing ABL application will work without change. When executing against a converted multi-tenant database, all existing ABL code that accesses database objects is implicitly tenant aware, which means that it accesses database objects that are shared (the default) or that belong to the tenancy of the connected user (if you have multi-tenant-enabled any database objects).

Super tenant users initially run an existing ABL application as the default regular tenant, which means they access only shared data and data owned by the default tenant. To make full use of super tenancy, which allows access to all the data in a multi-tenant database regardless of tenant ownership, ABL provides new features that you can add to existing applications or use to build new applications. These features allow a super tenant user to query and update the data owned by any one tenant, or group of tenants, defined in the database.

If you plan to create tenant groups for multi-tenant tables, or use any shared (non-multi-tenant) tables, and you also are considering the use of multi-tenant sequences, you need to plan carefully because of how multi-tenant sequences interact with any tables that are shared by multiple tenants.

For details, see the following topics:

- Running applications in a multi-tenant database environment
- Coding for super-tenant access
- Using multi-tenant-enabled sequences
- Managing access to tables with groups

Running applications in a multi-tenant database environment

The primary requirements to run an application against a multi-tenant database is to configure one or more tenants in the database, and to configure at least one domain per tenant in which application users can authenticate when they login to your application.

Existing applications

For a converted multi-tenant database, any existing application only requires the default tenant in order to execute as originally designed, and you typically add additional regular tenants to expand the application to run with totally separate sets of users, again, with little or no change to the application code.

If you already have domains defined in a converted database in order to sign on users using a client-principal object, and you configure each of these existing domains with a tenant, your application might work without any further changes depending on how you initialize attributes of the client-principal. Assuming these attributes are set correctly, existing calls to the `SET-DB-CLIENT` function or the `SET-CLIENT( )` method (on the `SECURITY-POLICY` system handle) will set the user tenancy along with the user identity of any multi-tenant database connection.

If your user login procedure relies entirely on the User ID (`-U`) connection parameter to authenticate database access (at startup or using the `CONNECT` statement), as long as you ensure that users provide a user ID that includes their assigned user name and domain name in the form, *`user-name@domain-name`*, and the domain is configured for an authentication system (such as the `_User` table accounts) to which OpenEdge can authenticate database connections, your application might also work without any further changes. The main thing is to ensure that your application sets database connection identity in a manner that authenticates users with a tenancy defined in your database.

You can design a number of different approaches (authentication models) to manage user access to OpenEdge databases. For more information on authenticating users in both multi-tenant and non-multi-tenant environments, see *Introduction to Identity Management* and Application Security on page 97.

Super-tenant applications

Note that applications written prior to OpenEdge Release 11 have no concept of tenancy as defined by OpenEdge. Virtually all code written for them works as of Release 11 with whatever tenancy is defined for the current database connection identity. To take advantage of super tenancy, you need to use the `SET-EFFECTIVE-TENANT` built-in function to acquire a new effective regular tenancy prior to executing an existing application, or write new applications or new modules for existing applications that require super-tenant access.

Typically, new super-tenant applications are used to maintain or monitor multi-tenant data across tenants. One example might be in a Software as a Service (SaaS) environment, where the SaaS administrator needs to collect billing information for all application tenants, and to add or update billing notices for applicable tenants.

Coding for super-tenant access

A super-tenant user (*super tenant*, in short) has access to all the data of a multi-tenant database, including all tenant and shared data, according to the data authorization provided by the user's permission settings. Super-tenant access is enabled by configuring the user's security domain with the name of a super tenant. Data authorization settings work the same as for any multi-tenant database user, with permissions for individual tables and fields set based on the user and domain name.

A super tenant has no real tenancy of its own, but always has the effective tenancy of a regular tenant. The initial tenancy of any super-tenant user is the default tenant. So, when initially logged in, a super-tenant user has implicit access to both default tenant and shared database objects. To support super-tenant access to multi-tenant database objects, ABL provides features to allow a super tenant to query and update database objects that belong to one or more of the regular tenants defined in the database. In addition, ABL allows a super tenant to identify the tenant and tenant group (if any) to which any given multi-tenant database record belongs.

These features, taken together, allow a super tenant to perform any run-time action on database objects, regardless of tenancy, and limited only by the data authorization permissions that are defined for a given super-tenant user. For example, if the super-tenant user identity has only can-read permissions on a given multi-tenant table, the super tenant can read instances of the table owned by all tenants, but cannot update any instances of the table.

The following sections provide an overview of OpenEdge super tenant capabilities in ABL.

- Setting effective tenancy on page 191
- Tenancy identification on page 192
- Super-tenant queries on page 194
- Super-tenant data updates on page 199
- Super-tenant sequence access on page 200

Setting effective tenancy

A super tenant can change effective tenancy for a given database connection with the ABL built-in function, `SET-EFFECTIVE-TENANT`. This function allows the super tenant to implicitly query data for the effective tenant without having to set a new user identity for the connection. All queries with this effective tenancy return only shared data and data owned by the effective tenant.

Once a super tenant has set an effective tenancy, the super tenant has all the same behaviors as the regular tenant for actions such as reading, creating, updating, and deleting data.

This is the syntax of the `SET-EFFECTIVE-TENANT` function:

Syntax

```
SET-EFFECTIVE-TENANT ( tenant-expression [ , database-name ] )
```

Where *`tenant-expression`* evaluates to the tenant name or ID of a valid regular tenant, and *`database-name`* evaluates to the logical name or alias of a connected database that defines the specified tenant. If you do not specify *`database-name`*, and multiple databases are connected, the function raises an error. The function returns logical, `TRUE` if successful, and raises an error if not successful.

The purpose of this function is to allow a super tenant user to act as if it is a particular regular tenant without having to become that tenant by authenticating a new user identity. It creates an *effective tenancy* which is distinguished from the real tenancy of the user's identity, which in this case is a super-tenancy. If a user identity with regular tenancy executes this function, the specified tenant must match the user's real tenancy, and if not, the function raises an error. If a super tenant executes this function and *`tenant-expression`* is not a valid regular tenant, the function also raises an error.

The scope of SET-EFFECTIVE-TENANT ends when either an authentication operation sets the database connection to a new identity with a different real tenancy or another invocation of SET-EFFECTIVE-TENANT sets a new effective tenancy. The difference between setting and effective tenant and changing real tenancy is in how OpenEdge handles current buffers and cursors.

When you execute SET-EFFECTIVE-TENANT, OpenEdge does **not** clear buffers and cursors with existing tenant information from any previous SET-EFFECTIVE-TENANT call. Also, an UNDO does not undo the effective tenancy set using SET-EFFECTIVE-TENANT. In this way, a super tenant can retrieve records for more than one tenant by changing effective tenancy. So, for example any existing cursor in a FOR EACH statement or query remains unaffected. However, when you set a new user identity, with or without a new tenancy, OpenEdge cleans up all buffers and cursors for the new user identity.

Caution: If you want to remove all record of super-tenant activity, you must reset the user identity for the connection instead of setting a new effective tenancy.

Note: If a super tenant does invoke SET-EFFECTIVE-TENANT to change effective tenancy within a FOR EACH block or during an existing query, the next records read by the cursor are still read as the original effective tenant, and if written, are written back to that same tenant. Any new queries or record finds after the current query ends, operate under the new effective tenancy.

Alternatively, when a super tenant needs records for more than one tenant they can use the TENANT-WHERE option of the record phrase to retrieve all tenant records without having to change their effective tenancy for each tenant.

Tenancy identification

Because a super-tenant user can access data for multiple regular tenants of a multi-tenant database, the first thing they often need to do is to identify the tenants of a connected database. Or they might need to identify the tenancy of specific data that they query across multiple tenants of the database, perhaps in order to identify the effective tenancy setting to more conveniently process the data for a given tenant.

OpenEdge identifies tenancy by:

- **Tenant name** — A name you assign when you configure the tenant in a multi-tenant database, or "Default" for the name of the permanently configured default tenant in every multi-tenant database.
- **Tenant ID** — A database-unique, integer ID that OpenEdge assigns when you first create the tenant in a database. The values for tenant IDs have standard ranges, depending on the type of tenant:
  - Default (regular) tenant ID — Always zero (0)
  - All other regular-tenant IDs — Positive integers
  - All super-tenant IDs — Negative integers

  Testing on this value allows you to select on multiple tenants by their tenant type.

The following sections describe various tools available for identifying tenancy for various contexts.

- Identifying database connection tenancy on page 193
- Identifying database record tenancy on page 193
- Identifying tenancy in temp-tables on page 193
- Identifying the tenants defined in a database on page 194

Identifying database connection tenancy

A super tenant can return information about database connection tenancy using these ABL built-in functions:

- **IS-DB-MULTI-TENANT function** — Returns TRUE if a connected database is multi-tenant enabled.
- **TENANT-NAME and TENANT-ID function** — Returns the tenancy associated with a current database connection identity.
- **TENANT-NAME-TO-ID function** — Returns the tenant ID, given the tenant name associated with a current database connection identity.
- **GET-EFFECTIVE-TENANT-NAME and GET-EFFECTIVE-TENANT-ID function** — Returns the current effective tenancy for a connected database.

Note: The GET-EFFECTIVE-TENANT-* functions return the most recent **effective** tenancy for a database connection, where as the corresponding TENANT-* functions return the **real** tenancy of the current database **connection identity**. The effective tenancy for a database connection can be set using the SET-EFFECTIVE-TENANT function (see Super-tenant queries on page 194), where as the **real** tenancy for the connection identity can **only** be set by applying a user authentication or single sign-on (SSO) operation to the connection. These two tenancies can be different, with the most recently set tenancy overriding the other for purposes of effective tenancy when accessing the database.

Note: You cannot use the TENANT-NAME or TENANT-ID function in the WHERE option of the record phrase. Instead, you can use these functions in the TENANT-WHERE option of the record phrase. For more information, see TENANT-WHERE option of the record phrase on page 194.

Identifying database record tenancy

A super tenant can get information about the tenancy of a database record using these ABL built-in elements:

- **IS-MULTI-TENANT attribute on a buffer handle** — Returns TRUE if the current record referenced by the specified buffer handle is in a multi-tenant table.
- **BUFFER-TENANT-NAME and BUFFER-TENANT-ID function** — Returns the tenancy of the current record in the specified buffer.
- **BUFFER-TENANT-NAME and BUFFER-TENANT-ID attribute on a buffer handle** — Returns the tenancy of the current record referenced by the specified buffer handle.

Note: You cannot use the BUFFER-TENANT-NAME or BUFFER-TENANT-ID functions or attributes in the WHERE option of the record phrase **if** these functions and attributes work on the **same** record buffer that is specified by the record phrase. Instead, you can use these functions and attributes in the TENANT-WHERE option of the same record phrase. For more information, see TENANT-WHERE option of the record phrase on page 194.

Identifying tenancy in temp-tables

Temp-tables have no inherent tenancy and do not automatically store the tenancy of multi-tenant data that might be queried to populate them. So, if your application relies on temp-tables that are populated from a multi-tenant database, especially in ProDataSets that are used to transport data through the layers of an application modeled on the OpenEdge Reference Architecture (OERA), or across the ABL sessions of any multi-tier or service-oriented application, you need to explicitly track the tenancy of the data, unless tenancy is implied in the application data itself.

Without a clear indication in the application data, you can add columns to explicitly identify the tenancy in any temp-tables that hold multi-tenant data.The maximum additions you might need to track the tenancy of data are columns sufficient to store the tenant name, tenant ID, and the name and path to the database where the data originates or is intended for update. If only one database is involved, you might need only one column to store the tenant name or ID.

You might also find other ways to identify the tenancy of data that is transported to and from its source, such as by using class or procedure-based objects instantiated to handle the data for specific tenants.

Identifying the tenants defined in a database

There are some applications where a super tenant might need to query the database directly for all the tenants that it defines, or where accessing multi-tenant data for a set of tenants is more conveniently done by querying the tenants from the database directly. The tenant records for all tenants configured in a database are stored in the `_Tenant` metaschema table, and the following fields identify the particular tenant defined by each `_Tenant` record:

- **`_Tenant-Name`** — A character field containing the tenant name
- **`_TenantID`** — An integer field containing the tenant ID

You can then query these fields to return the names and IDs of all tenants defined in the database without reference to database connection or record tenancy.

Note: You can also identify the groups defined for all tables with groups, and their tenant membership. For more information, see Managing access to tables with groups on page 202.

Super-tenant queries

With can-read access, a super-tenant user can query data for any one or a combination of tenants using the following basic ABL query mechanisms:

- **`TENANT-WHERE` option of the record phrase** — Adds tenancy criteria to the data in a query independent of the current effective tenancy. Given the specified multi-tenant tables and joins in a query, this option can be used to qualify the tenancy of each individual record buffer, including record instances from multiple tenants. If multiple specified tenants are in a tenant group, the associated `SKIP-GROUP-DUPLICATES` option skips all but the first tenant record in any group.
- **`REPOSITION` statement and `REPOSITION-TO-ROWID( )` method on a query handle** — Provides an option to reposition to a query row owned by a specified tenant.

TENANT-WHERE option of the record phrase

To allow a super tenant the ability to run a single query that retrieves data for multiple tenants, OpenEdge provides the `TENANT-WHERE` option. When used in `FOR EACH` statements and query phrases, a super tenant can retrieve records for multiple tenants regardless of their effective tenancy (see Setting effective tenancy on page 191). This is the syntax of the `TENANT-WHERE` option in the record phrase:

Syntax

```
{ record [ field-list ] } [ OF table ] [ WHERE expression ]
[ TENANT-WHERE tenancy-expression [ SKIP-GROUP-DUPLICATES ] ]
[ USE-INDEX index | TABLE-SCAN ]
[ other-existing-syntax ]
```

Where *tenancy-expression* is a relational expression, or logical combination of relational expressions, that test a valid tenant name or ID in the database that contains the *record*. Note that a WHERE option *expression* cannot involve the TENANT-ID and TENANT-NAME function or the BUFFER-TENANT-ID(*buffer*), BUFFER-TENANT-NAME(*buffer*), *buffer-handle*:BUFFER-TENANT-ID, and *buffer-handle*:BUFFER-TENANT-NAME functions and attributes if *buffer* or *buffer-handle* reference the **same** buffer as *record*; otherwise, ABL raises an error. Instead, you can use such expressions in *tenancy-expression* of the TENANT-WHERE option. However, the buffer functions and attributes can be used in expressions of the WHERE option if the specified buffers are **not** the same as *record*.

Note: Code that contains the TENANT-WHERE option can only be executed by a super tenant. However, any user can compile this code. If a regular tenant (including the default) attempts to run with this option, the AVM raises an error.

Note also that the TENANT-ID and TENANT-NAME functions in a TENANT-WHERE option expression must not include the optional database name parameter, because they must retrieve the tenancy of the same database connection where *record* is retrieved.

Although you can (and often need to) involve these functions and attributes in relational expressions of the TENANT-WHERE option, the remainder of each relational expression must involve only elements that are outside the current *record* buffer scope. For example, the TENANT-WHERE expressions in the following FOR EACH fragments are valid, assuming that cTenancy and iTenancy are character and integer variables, respectively:

```
FOR EACH Customer TENANT-WHERE BUFFER-TENANT-NAME(Customer) = cTenancy ...

FOR EACH Customer TENANT-WHERE BUFFER-TENANT-ID(Customer) > iTenancy ...
```

However, the following FOR EACH fragment raises a syntax error:

```
FOR EACH Customer TENANT-WHERE BUFFER-TENANT-ID(Customer) > Customer.tenid. ...
```

The syntax error occurs because the tenancy decision has to be made **before** any Customer records can be accessed at all. So, there is no way to get tenant criteria out of the Customer record, such as Customer.tenid.

If you use a TENANT-WHERE option in a join, you can have only one TENANT-WHERE option at only **one** level of the join. In other words, the option can appear in only **one** record phrase of a query with multiple record phrases. This is required so that when the query is executed for each tenant in succession, the entire sub-tree of related records are from the same tenant.

Also note that use of the BUFFER-TENANT-NAME and BUFFER-TENANT-ID functions and attributes in the TENANT-WHERE option are shortcuts for the _Tenant-Name and _TenantID fields in the _Tenant metaschema table, if the buffer referenced is the same as the *record* buffer in the record phrase. The following two TENANT-WHERE phrases are equivalent:

```
FOR EACH Customer TENANT-WHERE BUFFER-TENANT-NAME(Customer) BEGINS "E":
FOR EACH Customer TENANT-WHERE _Tenant._Tenant-Name BEGINS "E":
```

In another example, you can use the SET-EFFECTIVE-TENANT function instead of using TENANT-WHERE, as in the following example:

```
FOR EACH _Tenant WHERE _Tenant._Tenant-Name BEGINS "E":
  SET-EFFECTIVE-TENANT(_Tenant._Tenant-Name).
  FOR EACH Customer:
. . .
```

Note:

The *tenancy-expression* in the TENANT-WHERE option is used to define an index scan on the _Tenant table. The tenant records selected by this scan are share locked by the AVM. If the AVM cannot get the share lock within 60 seconds, the query raises error and aborts execution. Once the record is share locked, the share lock is held until the scan of the tenant's data is completed, at which point the lock is released. The share lock is released even if the TENANT-WHERE query executes within a transaction.

Other ABL elements described in these topics wait on a share lock when reading the _Tenant table to obtain tenant information. For more information on the process of holding and releasing the share lock for each element, see the appropriate reference entry in *ABL Reference*.

SKIP-GROUP-DUPLICATES option of TENANT-WHERE

If a multi-tenant table is configured with a tenant group, a single instance of the table is shared by all tenants in the group (see *Introduction to Database Multi-tenancy*). This means that if a super tenant queries the table for all tenants in the group, the query returns exactly the same records once for each tenant in the group.

For example, if two or more tenants are in a group configured for the Customer table, the following example displays the same Customer data for the group multiple times—once for every tenant in the group:

```
FOR EACH Customer TENANT-WHERE TENANT-ID >= 0:
  DISPLAY CustNum.
END.
```

Because the group data is common to all the tenants in the group, you see that same record once when the TENANT-WHERE processes the first member of the group, once again when the TENANT-WHERE processes the second member of the group, and so on. You may want this if you are referring to other tables in the FOR EACH statement that do not have the same grouping, since they would be missed otherwise. But if you have a very simple case like the example above, with no join or other references to groups that do not have the same tenant membership, you probably want to skip the duplicate data. In that case you can use the SKIP-GROUP-DUPLICATES option.

This example only displays group data for the first member of the group, and skips tenants that belong to the same group that has already been processed:

```
FOR EACH Customer TENANT-WHERE TENANT-ID >= 0 SKIP-GROUP-DUPLICATES:
  DISPLAY CustNum.
END.
```

This is not the default behavior, because you typically do have references to other tables in a `FOR EACH` block or in a query.

Query examples without groups

Following are some super-tenant query examples. They all assume that `Customer` is a multi-tenant table with no groups defined.

The following example shows a simple query.

Super-tenant query for no explicit tenant

```
FOR EACH Customer WHERE cust-num < 100.
```

If the code in the above example is executed by a regular tenant with no `Customer` tenant group, the user sees only their own customers with `cust-num < 100`.

If the code in the above example is executed by a default tenant or a super tenant that has not executed `SET-EFFECTIVE-TENANT`, the user sees only the default tenant's customers with `cust-num < 100`. If there is no default partition, the user will see no data.

If the code in the above example is executed by a super tenant that has used `SET-EFFECTIVE-TENANT` to acquire an effective tenancy, the user sees only that tenant's customers with `cust-num < 100`.

The following example adds a simple `TENANT-WHERE` option that selects one tenant for the simple query.

Super-tenant query for one tenant

```
FOR EACH Customer WHERE cust-num < 100 TENANT-WHERE TENANT-ID = 3.
```

If the code in the above example is executed by a regular tenant, OpenEdge raises a run-time error.

If the code in the above example is executed by a super tenant, the user sees customers with `cust-num < 100` for the tenant whose tenant ID is 3. The super tenant sees the same result regardless of their effective tenancy.

The following example expands the `TENANT-WHERE` option to select multiple tenants based on compound conditions.

Super-tenant query for multiple tenants qualified by name and ID

```
FOR EACH Customer WHERE cust-num < 100
  TENANT-WHERE TENANT-NAME BEGINS "lowes" AND TENANT-ID >= 3.
```

The code in the above example behaves like the **Super-tenant query for one tenant** example, except instead of only output for the one tenant whose tenant ID is 3, the output includes all tenants where the tenant name begins with `"lowes"` and the tenant ID is 3 or greater.

The following example executes legacy code for every regular tenant in the database.

Super-tenant query using the _Tenant table

```
FOR EACH _Tenant WHERE _Tenant._TenantID >= 0:
  SET-EFFECTIVE-TENANT(_Tenant._TenantID).
  RUN legacy.p. /* May refer to many tables with FINDs/CREATEs/FOR EACHs/etc */
END.
```

The code in the above example allows a super tenant to execute the legacy code on behalf of all the tenants in the `_Tenant` table without reference to a specific tenant.

The following example reads separate buffers, each with its own effective tenancy.

Super-tenant queries on buffers with various effective tenancies

```
DEFINE BUFFER bufCust FOR Customer.
 SET-EFFECTIVE-TENANT(1).

FIND FIRST Customer.
SET-EFFECTIVE-TENANT(3).
FIND FIRST bufCust.
```

The code in the above example shows how a super tenant might read data simultaneously from various tenants by defining different buffers for the same multi-tenant table and reading each one with a different effective tenancy. The code in this figure reads the first `Customer` for tenant 1 into the `Customer` buffer, and the first `Customer` for tenant 3 into the `bufCust` buffer. If the super tenant later becomes the effective default, or any other regular tenant, they can continue to update the previously populated buffers for their corresponding tenants (assuming appropriate can-* permissions). For more information on multi-tenant updates, see Super-tenant data updates on page 199.

Query examples with groups

By adding the `SKIP-GROUP-DUPLICATES` option, the `TENANT-WHERE` option on a `FOR EACH` automatically guarantees that duplicate records that might otherwise be displayed for tenants that are in the same tenant group are only displayed once per group. The following examples illustrate how queries with a `TENANT-WHERE` option handle tenant-group data. They all assume that `Customer` is a multi-tenant table with at least one tenant group defined.

The following example is a simple query with a simple `TENANT-WHERE` option that selects one tenant in a group.

Super-tenant query for one tenant in a group

```
FOR EACH Customer WHERE cust-num < 100 TENANT-WHERE TENANT-ID = 3.
```

If the code in the above example is executed by a regular tenant, OpenEdge raises a run-time error.

If the code in the above example is executed by a super tenant, the user sees customers with `cust-num < 100` for the tenant whose tenant ID is 3, or if tenant 3 is part of a group defined for `Customer`, they see the whole group's customers with `cust-num < 100`. The super tenant sees the same result regardless of their effective tenancy.

The following example shows a simple query with an compound `TENANT-WHERE` option that selects for multiple tenants in a group, both excluding and including the option to skip group duplicates.

Super-tenant query on multiple tenants in groups

```
FOR EACH Customer WHERE cust-num < 100 TENANT-WHERE
  TENANT-NAME BEGINS "lowes" and TENANT-ID >= 3.

FOR EACH Customer WHERE cust-num < 100 TENANT-WHERE
  TENANT-NAME BEGINS "lowes" and TENANT-ID >= 3
SKIP-GROUP-DUPLICATES.
```

The first `FOR EACH` statement in the above example behaves like the **Super-tenant query for one tenant in a group** example, except instead of only output for the one tenant whose tenant ID is 3, the output includes all tenants where the tenant name begins with `"lowes"` and the tenant ID is 3 or greater, and in any groups that might apply. In this case, because there are multiple tenants in the output, it can happen that the same group data is returned more than once. If the `SKIP-GROUP-DUPLICATES` option is used, as in the second `FOR EACH`, the `TENANT-WHERE` mechanism detects this and skips the output for tenants belonging to a group (or set of groups in a join) that has already been seen.

The following example executes legacy code that references tables with groups for every regular tenant in the database.

Super-tenant query using the _Tenant table involving groups

```
FOR EACH _Tenant WHERE _Tenant. _TenantID >= 0:
  SET-EFFECTIVE-TENANT(_Tenant. _TenantID).
  RUN legacy.p. /* May refer to many tables with FINDs/CREATEs/FOR EACHs/etc */
END.
```

The code in the above example allows a super tenant to execute the legacy code on behalf of all the tenants in the `_Tenant` table without reference to a specific tenant.

Because multiple tenants for the same group will produce multiple invocations of `legacy.p` for the group, this might cause problems. The `legacy.p` procedure might have many table references, with completely different group configurations for each one, in which case you would really want to have `legacy.p` run for each of these tenants anyway, despite the group situation.

And if all the tables in `legacy.p` share the same group configuration, you can then, for example, test the data for which group identity programmatically using the `_Partition-Set` metaschema table to avoid the repetition. For more information on testing for group identity, see Managing access to tables with groups on page 202.

Super-tenant data updates

ABL allows a super tenant to create (with can-create access) and update (with can-write or can-delete access, as appropriate) database records for specified tenants using the following elements:

- **`CREATE` statement and `BUFFER-CREATE( )` method on a buffer handle** — For a multi-tenant table, these ABL elements provide an option for a super tenant to create a database record for a specified tenant, regardless of the effective tenancy. Without this option, these ABL elements create database records owned by the effective tenant.
- **All record update statements, such as `ASSIGN` , `SET` , `UPDATE` , `DELETE` , etc.** — Regardless of the current effective tenant, ABL remembers the tenancy of all records created in or retrieved from a multi-tenant database. When a super tenant updates database records created for multiple tenants or queried for multiple tenants, ABL ensures that all these record instances are written to or deleted from the table instances owned by the corresponding tenants.

The universal rule is that if a record in a database buffer is owned by a given tenant or tenant group, ABL writes any updates to that record only for the same tenant or tenant group, regardless of the user's effective tenancy.

Note: If data you want to change in multi-tenant tables is updated from temp-tables, or any other internal or external data store outside of the OpenEdge multi-tenant database, you must track the tenancy of any data that is not maintained in database buffers. For more information, see Identifying tenancy in temp-tables on page 193.

Super-tenant sequence access

Similar to multi-tenant tables, a super tenant can access a multi-tenant sequence owned by a particular tenant using the following functions:

- **`CURRENT-VALUE` and `DYNAMIC-CURRENT-VALUE` function** — Provides an option to return the current value of the multi-tenant sequence for the specified tenant.
- **`NEXT-VALUE` and `DYNAMIC-NEXT-VALUE` function** — Provides an option to return the next value of the multi-tenant sequence for the specified tenant.

However, note that if you configure, or ever intend to configure, tables with tenant groups or shared tables in a multi-tenant database, certain uses of a multi-tenant sequences can have unintended consequences, both for regular and super-tenant users. The following section describes these potential issues.

Using multi-tenant-enabled sequences

You can create a sequence that is multi-tenant enabled. However, you cannot convert an existing non-multi-tenant sequence to be a multi-tenant sequence. You need to delete the sequence and recreate it in order to make it multi-tenant-enabled.

If a sequence is multi-tenant enabled, a separate current value, for the same sequence of values, is maintained for each regular tenant in the database. The built-in sequence statements and functions `NEXT-VALUE`, `CURRENT-VALUE`, `DYNAMIC-NEXT-VALUE`, and `DYNAMIC-CURRENT-VALUE` contain an optional parameter for specifying a particular tenant value, primarily for use when coding for a super tenant0

Syntax

```
[ NEXT-VALUE | CURRENT-VALUE ] ( sequence-name  [ , logical-dbname
  [, tenant-id ] ]) [ = expression ]

[ DYNAMIC-NEXT-VALUE | DYNAMIC-CURENT-VALUE ] ( sequence-name-expression
  , logical-dbname-expression  [, tenant-id ] ) [ = expression ]
```

`tenant-id`

An integer expression that only has meaning for a multi-tenant-enabled sequence and that evaluates to a valid tenant ID for a regular tenant. A super tenant can specify any regular-tenant ID. If a super tenant does not specify this argument, the statement or function applies to the effective tenant.

Typically, a regular tenant never uses this argument. If a regular tenant does use the *`tenant-id`* argument, the specified tenant must be the same as the tenant associated with the database connection. Otherwise, the AVM raises an error.

If the sequence is **not** enabled for multi-tenancy, and the *tenant-id* argument is specified, the argument is ignored.

For more information on using the typical arguments with these statements and functions, see Sequences on page 81.

A regular-tenant user, like the user of a non-multi-tenant database, typically accesses these statements and functions without specifying a tenant ID, because they can only access the sequence instance with their tenancy. However, database triggers can execute for both regular and super tenants. So, appropriately specifying the tenant ID in database triggers ensures that the correct sequence is used in all cases. The following example demonstrates the database create trigger that uses the *tenant-id* argument:

```
TRIGGER PROCEDURE FOR CREATE OF Customer:
  ASSIGN Customer.CustNum
    = NEXT-VALUE( NextCustNum, sports2020, BUFFER-TENANT-ID( Customer ) ).
END.
```

In the above example, `Customer` is a multi-tenant table. Use of the `BUFFER-TENANT-ID` function ensures that for a super tenant, the trigger increments the customer number using the multi-tenant `NextCustNum` sequence for the tenant that owns the current `Customer` record. For more information on the `BUFFER-TENANT-ID` function, see Managing access to tables with groups on page 202.

Note that because every regular tenant has its own instance of a multi-tenant sequence, the values that the sequence generates are unique only for multi-tenant table instances owned by that tenant. If the sequence is used to generate values for a shared table, those values will not be unique because every tenant generates the same values, and there is no automatic way to tell which tenants have generated the duplicate values. This means that care must be taken when deciding whether to make a sequence multi-tenant and where to use it.

Using multi-tenant sequences with shared and group-configured tables

Shared and tables with groups can be useful, but using them with sequences requires some care. The challenge has to do with the fact that shared tables and tables with tenant groups are shared by some or all tenants of the database, where as multi-tenant sequences have no groups and are never shared, but are always separately owned by each regular tenant in the database.

This means that for each multi-tenant sequence that you define, every tenant that executes a sequence function generates the same set of values. The tenant can then assign these values to a shared or group-configured table, but with the following problems:

- If the field requires a unique value, as for a primary index, the AVM raises an error when any tenant tries to assign an existing value previously assigned by another tenant.
- If the field represents a foreign key, queries on that foreign key with the same value can return multiple records from different tenants. However, information about which tenant owns which record is lost within the group or shared index on that foreign key.

As a result, to use sequences effectively with shared and group-configured tables requires certain restrictions. The following example demonstrates some of the problems with sequence interactions with shared and group-configured tables.

Suppose the `Customer` table in a database is multi-tenant-enabled and does not participate in any group. The `Customer.cust-num` key is generated by a sequence that has been made incorrectly multi-tenant. The `Order` table also contains `cust-num` as a foreign key and it participates in a group with tenants 1 to 3. A `Customer` record is created by tenants 1, 2, and 3. All three `Customer` records get the `cust-num` value `1` because each tenant has its own instance of the multi-tenant sequence, as well as its own instance of the `Customer` table. However, as tenants 1, 2, and 3 create orders for customer 1, you cannot identify the tenant that owns each `Order` record written for customer 1, where `Order.cust-num = 1`, because all three tenants in the group share the same `Order` table instance.

When you run the following query for tenant 2, it displays the orders for customer 1 from all three tenants that share the `Order` table:

```
FOR EACH Customer, EACH Order OF Customer:
  DISPLAY Order.cust-num Order.order-num.
END.
```

Because there is only one index in this group table containing `cust-num` and `order-num` for the entire group of orders, all orders for all `Customer` records with `cust-num = 1` are treated the same for all three tenants.

Restrictions for using groups and sequences

To avoid such problems, you must follow one of the following rules for using sequences with shared and group-configured tables:

- Ensure that all related tables have the same group definitions, with the same tenants defined as members of a given group defined in each table.
- Have grouping at higher levels of join but not at lower levels.
- Generate all foreign keys used in group or shared tables from a shared sequence, guaranteeing that each tenant will generate a unique value.

The best practice while working with group or shared tables is to always use a shared sequence to generate any field values to be used as a foreign key.

For more information on managing sequences and multi-tenant tables with groups, see Managing access to tables with groups on page 202.

Managing access to tables with groups

Groups have an identity, similar to tenancy for tenants, which is identified by a group name and integer ID.

When creating new group records, you might chose to implement separate shared sequences to assign key values for each group defined for a table. In the CREATE trigger, where you typically assign the key values to a record, you can test the group identity for the record in order to determine the sequence used to assign the key value. You can return the group identity for any record in a group table using the following ABL elements:

- **`BUFFER-GROUP-NAME` and `BUFFER-GROUP-ID` function** — Returns the the tenant group identity that owns the current record in the specified buffer.
- **`BUFFER-GROUP-NAME` and `BUFFER-GROUP-ID` attribute on a buffer handle** — Returns the tenant group identity that owns the current record referenced by the specified buffer handle.

You might want to identify all the groups defined for a table with groups, and the tenant membership of each group. The `_Partition-Set` metaschema table defines all the groups in a database, with the following fields useful for identifying related tables and tenant membership:

- **_Object-Number** — An integer field with the object number of the multi-tenant table for which this group is defined, and is the same value as the _File.File-Number field in the _File record that defines the table.
- **_PSetID** — An integer field with the group ID assigned by OpenEdge.
- **_PSet-Name** — A character field with the group name that you assign when defining a group for a multi-tenant table.

The _Partition-Set table is related to a _Partition-Set-Detail table by the _Object-Number and _PSetID fields. You can therefore use the following fields of _Partition-Set-Detail to identify the tenants that are members of a given group for a table:

- **_TenantID** — An integer field containing the tenant ID that identifies a tenant defined in the _Tenant table. For more information, see Identifying the tenants defined in a database on page 194.
- **_Partition-set.id** — An integer field containing the group ID that identifies a group defined in the _Partition-Set table.

4

ABL API for Multi-tenant and Table Partition Management

OpenEdge provides an API that provides access to multi-tenant and table partition entity data and operations typical of the Data Dictionary. The API can be used by developers who want or need to use ABL code to manage multi-tenant and table partition information. The API is provided as a set of ABL user-defined classes and interfaces for accessing database entities. The user-defined classes include a single API service class that handles all communication between the database and entity objects. The API provides user-defined interfaces to access all these entity objects, most of which the service class creates or retrieves for a given database using factory methods.

For a reference to the entity interfaces, see Database Administration Entity Interface Reference on page 707, and for a reference to entity interface collections, see Database Administration Entity Collection Interface Reference on page 773.

For details, see the following topics:

- Overview of the API
- Instantiating a DataAdminService
- Managing the service life cycle
- Managing transactions
- Maintaining multi-tenant and table partition entities
- Filtering and sorting
- Exporting and importing
- Exporting and Importing Partition Policies and Details

- Executing DataAdminService utilities
- Handling errors

Overview of the API

The purpose of the API is to allow ABL code to read and update multi-tenant and table partition related schema information. It is designed to provide access to the typical operations for configuring the tenants of a multi-tenant database, configuring the partitions of a horizontally partitioned table, and provides access to the schema information required to complete these operations. In addition to entity objects, the API supports various types of collection objects for managing multiple entity instances at one time.

Before attempting to use this API, you should be familiar with configuring multi-tenant databases and partitioned tables using the OpenEdge Data Dictionary, Database Administration Console in the OpenEdge Explorer and OpenEdge Management, and other database utilities. For more information, see the following manuals:

- *Introduction to Database Multi-tenancy*
- *Manage the OpenEdge Database*
- *Get Started with Database Multi-tenancy in OpenEdge Management*
- *Configure Database Multi-tenancy with OpenEdge Management*

Using this API, you can:

- Create, read, update, or delete authentication systems, domains, tenants, tenant groups, partition policies, indexes and users defined in the database.
- Read and create areas and extents, but you cannot update or delete them.
- Update multi-tenant information for existing tables and read information, including partition information, on table indexes and large object (LOB) fields provided as child collections
- Allocate table, index, and field partitions and edit the area for any partition; the creation and deletion of related schema elements happens automatically.
- Create, update, and delete sequences. Creating sequences is a multi-tenant schema operation, because you can only create a new sequence as a multi-tenant after deleting any existing shared sequence that you want to create as multi-tenant. Multi-tenant sequences also have a child collection that allows you to edit the current value for each tenant.

Other general schema objects cannot be created, deleted, or updated by the API.

Although the API does not support the direct schema creation, deletion, and update of tables, fields, and indexes, the API does support the loading of definition files (`.df`) and provides access to the same load options as the Data Administration utility as well as specific multi-tenant and table partition options. You can also conveniently access all partitions created as part of loading these definitions.

User identity management is important for multi-tenant management, and the API provides access to the permission information on tables and fields defined in the database. It also allows you to update the Security Administrators defined for an OpenEdge database by updating the `CAN-DO` function expression used in the database to specify user ID pattern lists for Security Administrators.

API packages, classes, and interfaces

The API provides ABL developers with the ability to write code to access and update the core schema without knowing the details of the core implementation. These implementation details are encapsulated by a set of ABL user-defined classes. In addition, the API provides access to most of these classes using a set of public interfaces that will continue to work in future OpenEdge versions without affecting previous application code that uses them.

These public interfaces are provided in the following packages:

- **`OpenEdge.DataAdmin`** — Interfaces for accessing all database entity (and related) objects (such as objects for areas, tables, tenants, and partitions) and all collections of these entity objects, and for accessing the API service class that handles database access for these entity objects and collections
- **`OpenEdge.DataAdmin.Lang.Collections`** — An interface that supports a general iterator for navigating the entity objects in all entity collections
- **`OpenEdge.DataAdmin.Util`** — Interfaces for accessing supported utility classes

---

Note: Interfaces defined in other packages are for OpenEdge use only. Some interfaces in the supported packages are also for OpenEdge use only. When in doubt, use only those interfaces described for access in this manual.

These supported packages also contain class implementations of all interfaces. These implementing classes are supported directly only for internal use because their constructors and members might change with each release. To minimize application code changes from release to release, always reference any API class using a public interface type that the class implements.

In addition, the API provides a set of data error classes in the `OpenEdge.DataAdmin.Error` package, and an error handler class for displaying the error messages thrown from the API. The API service class also supports properties and methods for managing all data errors raised by the service.

For a complete view of the API packages, you can access the `dataadmin` resource in the Class Browser view of Progress Developer Studio for OpenEdge.

Interface naming conventions

The API uses a standard naming convention for public interface names:

Syntax

```
IClassName [ Element | Collection ]
```

Where *`ClassName`* is the name of the implementing class for the interface. For entity objects, *`ClassName`* is the entity name. For example, the `IArea` interface is implemented by the `Area` class (for area objects) and the `IAreaSet` interface is implemented by the `AreaSet` class (for area collection objects). In the case of a few public super interfaces, which are implemented by several classes, *`ClassName`* is a more general name for the group of implementing classes and is appended with `Element` (for entity classes) or `Collection` (for entity collection classes). For example, `IDataAdminElement` is implemented by all entity classes and `IDataAdminCollection` is implemented by all entity collection classes. In addition, the API supports multiple kinds of entity collections, depending on the entity, and collection interfaces have a more detailed naming convention based on the kind of collection. For more information, see Collection interface naming conventions on page 774.

Creating and accessing objects with the API

When you use the API, you create or retrieve objects associated with entity interfaces using factory and retrieval methods of the API service class. In addition, child collection objects are returned using properties or methods of parent entity or collection objects. However, you must instantiate the API service and the following classes directly using the ABL `NEW` function (or related ABL class instantiation mechanisms):

- **`OpenEdge.DataAdmin.DataAdminService`** — The API service class, which you typically reference as an `OpenEdge.DataAdmin.IDataAdminService` interface type, and referred to generally as the DataAdminService (or service) throughout this manual
- **`OpenEdge.DataAdmin.RequestInfo`** — A class for specifying filtering criteria on child collections of requested entity objects, which you typically reference as an `OpenEdge.DataAdmin.IRequestInfo` interface type
- **`OpenEdge.DataAdmin.Util.DeallocateUtility`** — A utility class for deallocating partitions, which you typically reference as an `OpenEdge.DataAdmin.Util.IDataAdminUtility` interface type

---

Note:

These interfaces are intended for run-time usage. They might be extended with additional properties and methods with each release, and classes that implement an interface in this release will likely not compile in a future release. In addition, some interfaces define members that are supported for OpenEdge use only. These restricted members are identified in the description of the relevant interfaces.

Any public methods in implementing classes that are not defined in the corresponding interface might not be available in a subsequent release.

Each package also contains source code that is internal to the API service and not supported for use by applications. In general, use of the public interfaces protects against any accidental use of this source code.

However, very few of the implementing classes are defined as final. It is conceivable you might want to extend (inherit) these classes to resolve some requirement or fix a limitation. Any class extensions that you define are very likely to need changes when you install a new release.

API entities and entity service methods

The `IDataAdminstrationService` interface provides service methods that implement create, read, update, and delete operations against single schema elements, or a collection of schema elements, accessed through supported entity and entity collection interfaces. The following tables list the available service methods for accessing the basic entities and entity collections with these operations. The listed entity name is the implementing class name for each entity object and is also part of each entity method name, for example, "Area" as in the `NewArea( )` method. Corresponding entity collection methods have a similar method name using the plural of the entity name, for example in the `NewAreas( )` method.

All method parameters are passed as `INPUT` and are indicated in the method syntax by data type only. Method return types are shown only for methods that return entity or entity collection objects. All other entity service methods return a `LOGICAL` of `TRUE` when execution returns successfully.

The API supports additional entities (for example partitions) that you can access as child entity or collection objects using properties and methods of the basic entity objects that you access through the service.

The following table provides a summary of factory and create methods on the `IDataAdminstrationService` interface for entities and entity collections that you can create through the service.

Table 24: Multi-tenant and Table Partition entity factory and create methods

Entity name	Factory (New) methods	Create methods

Area	`IArea`
	`  NewArea( CHARACTER )`
	
	`IAreaSet`
	`  NewAreas( )`
		`CreateArea( IArea )`
		
		`CreateAreas( IAreaSet )`

AuthenticationSystem	`IAuthenticationSystem`
	`   NewAuthenticationSystem( CHARACTER )`
	
	`IAuthenticationSystemSet`
	`  NewAuthenticationSystems( )`
		`CreateAuthenticationSystem(`
		`  IAuthenticationSystem )`
		
		`CreateAuthenticationSystems(`
		`  IAuthenticationSystemSet )`

Domain	`IDomain`
	`  NewDomain( CHARACTER )`
	
	`IDomainSet`
	`  NewDomains( )`
		`CreateDomain( IDomain )`
		
		`CreateDomains( IDomainSet )`

Extent	`IExtent` `  NewExtent( )`	–

Sequence	`ISequence`
	`  NewSequence( CHARACTER )`
	
	`ISequenceSet`
	`  NewSequences( )`
		`CreateSequence( ISequence )`
		
		`CreateSequences( ISequenceSet )`

Tenant	`ITenant`
	`  NewTenant( CHARACTER )`
	
	`ITenantSet`
	`  NewTenants( )`
		`CreateTenant( ITenant )`
		
		`CreateTenants( ITenantSet )`

TenantGroup	`ITenantGroup`
	`  NewTenantGroup( CHARACTER )`
	
	`ITenantGroupSet`
	`  NewTenantGroups( )`
		`CreateTenantGroup(`
		`  ITenantGroup )`
		
		`CreateTenantGroups(`
		`  ITenantGroupSet )`

User	`IUser`
	`  NewUser( CHARACTER )`
	
	`IUserSet`
	`  NewUsers( )`
		`CreateUser( IUser )`
		
		`CreateUsers( IUserSet )`

Entity name	Factory (New) methods	Create methods

PartitionPolicy	`IPartitionPolicy NewPartitionPolicy( CHARACTER )`
	
	`IPartitionPolicySet NewPartitionPolicies( )`
		`CreatePartitionPolicy( IPartitionPolicy )`
		
		`CreatPartitionPolicies( IPartitionPolicyCollection )`

PartitionPolicyDetail	`IPartitionPolicyDetail NewPartitionPolicyDetail( CHARACTER )`	–

The factory (`New*( )`) methods return a new entity class instance (as the entity interface type) or an empty entity collection object (as the entity collection interface type), which you can update and pass to the corresponding `Create*( )` methods to commit the changes to the database.

The entity factory methods take a character key as input and return a new entity object with the passed key assigned to the associated key property (typically `Name` or `Id`). All other properties are set to their default values. The `NewExtent( )` method does not take any parameter and the service does not have a corresponding create method, because you add the extent object to an area object and committed it with the area object.

The collection factory methods have no parameter and return a new empty collection object. You can then add entity instances returned from the entity factory methods to the collection object and pass it to the corresponding `Create*( )` method for the collection to commit all the entities to the database in a single transaction. For more information on transactions, see Managing transactions on page 214.

For `IPartitionPolicyDetail`, a factory method that returns an empty collection does not exist, since the details collection always exists on the policy. Similarly, for IIndex, a factory method that returns an empty collection does not exist, since the a policy always consists an index.

Note:

The instances returned from the entity factory methods can also be added to existing collections retrieved with a `Get*( )` method, then committed with the corresponding `Update*( )` method in the service (see the following table). This is only necessary if the new instances need to be committed in the same transaction as other changes.

Similarly, the collections returned from the collection factory methods can also be added to existing collections retrieved with a `Get*( )` method and then committed with the corresponding `Update*( )` method in the service. This approach is only necessary if you need to ensure that instances imported from an external file are treated as new instances and gives an error if an instance with the same key already exists.

The following table provides a summary of read and update methods on the `IDataAdminstrationService` interface for entities and entity collections that you can read and update (if applicable) through the service.

Table 25: Multi-tenant and Table Partition entity read and update methods

Entity name	Read methods	Update methods

Administrator	IAdministrator
	  GetAdministrator( )
		UpdateAdministrator(
		  IAdministrator)

Area	IArea
	  GetArea(CHARACTER)
	
	IAreaSet
	  GetAreas( )
		UpdateArea(IArea)
		UpdateAreas(IAreaSet)

AuthenticationSystem	IAuthenticationSystem
	  GetAuthenticationSystem(
	  CHARACTER)
	
	IAuthenticationSystemSet
	  GetAuthenticationSystems( )
		UpdateAuthenticationSystem(
		  IAuthenticationSystem)
		
		UpdateAuthenticationSystems(
		  IAuthenticationSystemSet)

Domain	IDomain
	  GetDomain(CHARACTER)
	
	IDomainSet
	  GetDomains( )
		UpdateDomain(IDomain)
		
		UpdateDomains(IDomainSet)

Schema[7]	ISchema
	  GetSchema( )
	
	ISchema
	  GetSchemaChanges(CHARACTER
	)
		UpdateSchema(ISchema)
		
		UpdateSchemaChanges(
		  ISchema)

Sequence	ISequence
	  GetSequence(CHARACTER)
	
	ISequenceSet
	  GetSequences( )
		UpdateSequence(ISequence)
		
		UpdateSequences(ISequenceSet
		  )

Table	ITable
	  GetTable(CHARACTER)
	
	ITableSet
	  GetTables( )
		UpdateTable(ITable)
		UpdateTabless(ITableSet)

TablePermission	ITablePermission
	  GetTablePermission(CHARACTER
	  )
	
	ITablePermissionSet
	  GetTablePermissions( )
		UpdateTablePermission(
		  ITablePermission)
		
		UpdateTablePermissions(
		  ITablePermissionSet)

Entity name	Read methods	Update methods

Tenant	`ITenant`
	`  GetTenant( CHARACTER )`
	
	`ITenantSet`
	`  GetTenants( )`
		`UpdateTenant( ITenant )`
		
		`UpdateTenants( ITenantSet )`

TenantGroup	`ITenantGroup`
	`  GetTenantGroup( CHARACTER )`
	
	`ITenantGroupSet`
	`  GetTenantGroups( )`
		`UpdateTenantGroup(`
		`  ITenantGroup )`
		
		`UpdateTenantGroups(`
		`  ITenantGroupSet )`

User	`IUser`
	`  GetUser( CHARACTER )`
	
	`IUserSet`
	`  GetUsers( )`
		`UpdateUser( IUser )`
		
		`UpdateUsers( IUserSet )`

UserPermission	`IUserPermission`
	`  GetUserPermission( CHARACTER`
	`  )`
	
	`IUserPermissionSet`
	`  GetUserPermissions( )`
		–

PartitionPolicy	`IUserPermission`
	`  Get PartitionPolicy (`
	`CHARACTER )`
	`  Get PartitionPolicy ( INTEGER`
	`  )`
	
	`IUserPermissionSet`
	`  GetPartitionPolicies( )`
		`Update PartitionPolicy (`
		`IPartitionPolicy )`
		
		`UpdatePartitionPolicies(`
		`IPartitionPolicySet )`

The `Get*( )` methods return a single entity instance or a collection with entities that you can update, then pass to a corresponding `Update*( )` method to committed the changes to the database.

Most entity `Get*( )` methods take a character key as input (which typically maps to the `Name` property value) and return an existing entity object. Some entity `Get*( )` methods are overloaded to take an integer key (which typically maps to the `Id` property value) as an alternative to the character parameter shown in the above table.

For entity collection `Get*( )` methods, the service also provides overloadings with an `OpenEdge.DataAdmin.IRequestInfo` parameter to provide filter information for the retrieved collection, as well as any child collections. For more information, see Filtering and sorting on page 227.

The following table provides a summary of delete methods on the `IDataAdminstrationService` interface for entities and entity collections that you can delete through the service.

[7] The schema service methods represent a special case, because of their operation affects the entire database schema. For more information, see Schema (reading, loading, and updating) on page 219.

Table 26: Multi-tenant and Table Partition entity delete methods

Entity name	Delete methods
AuthenticationSystem	`DeleteAuthenticationSystem( CHARACTER )`
Domain	`DeleteDomain( CHARACTER )`
Sequence	`DeleteSequence( CHARACTER )`
Tenant	`DeleteTenant( CHARACTER )`
TenantGroup	`DeleteTenantGroup( CHARACTER )`
User	`DeleteUser( CHARACTER )`
PartitionPolicy	`DeletePartitionPolicy( CHARACTER )`

The `Delete*( )` methods take a single character key as input and delete the corresponding entity from the database without any warning or prompt. The methods throw an error if an entity with the specified key does not exist.

Note: You can also remove multiple entity objects from existing collections retrieved with an entity collection `Get*( )` method, then commit the collections with the corresponding `Update*( )` method. This approach allows you to make many deletions, along with other changes, to be committed in a single transaction. For more information on transactions, see Managing transactions on page 214.

Instantiating a DataAdminService

You can instantiate a DataAdminService for one database. The service methods, entity objects, and entity collection objects for that service instance all apply to that single database.

The following code shows how to instantiate the service for a connected database by passing the logical database name to the constructor:

```
using OpenEdge.DataAdmin.DataAdminService.

define variable service as DataAdminService no-undo.
service = new DataAdminService("sportsmt").
```

You can also instantiate the service by specifying the full physical path and database name in the constructor as follows:

```
service = new DataAdminService("C:\OpenEdge\WRK\db","sportsmt").
```

The database connects only to multi-user if there is a database server available.

Note: The service works best with local connections. Execution of the `DataAdminUtility` classes (with the `ExecuteUiltity()` method) does not work with a remote connection.

Managing the service life cycle

It is your responsibility to delete the service to release memory when the service is no longer needed. The service and the classes retrieved by the service are not automatically garbage collected, because of the complexity of object references among them. The service needs to remain alive while working on any object retrieved by the service. Deleting the service will also delete all objects retrieved by the service. For example:

```
define variable service as DataAdminService no-undo.
service = new DataAdminService("sportsmt").
/* API operations that instantiate or retrieve objects */
. . .
delete object service.
```

The service can be started and stopped per request or function or kept alive during the session. The service can accumulate a rather large amount of objects and data over time and is not particularly expensive to start and it is likely to be more efficient to bundle the start and stop of the service with each operation rather than keeping the same service alive for multiple operations. On the other hand, there is some overhead involved with each first reference to a type of entity, so multiple operations on the same type of instances will be more efficient using the same service. Instances realized through the `New*( )` or `Get*( )` methods in the service have to be passed to corresponding `Create*( )` or `Update*( )` methods of the same service instance.

One can start multiple instances of the service simultaneously. The intention is that one service is started per connected database. It is possible to start many services per database, but this will cause stale data and schema locks if more than one service operates on the same data. This can also happen if the service operates on entities that are also being updated by OpenEdge data administration tools.

Note: Passing instances realized through `New*( )` or `Get*( )` methods in the service to `Create*( )` or `Update*( )` methods of another service is unsupported. Such operations normally throw an error, but are unsupported even if they work without error: they are likely to throw an error in a future release.

Managing transactions

The `Update*( )`, `Create*( )`, and `Delete*( )` service methods normally commit the changes for an object in a single transaction. As shown above, the `Create*( )` and `Update*( )` methods support both a single entity and a collection of entities. A collection passed to an `Update*( )` service method can include new entities, changes to entities, and entity deletions.

The transaction includes changes to entities in any of the child collections of the entities that are passed to the service method. However, the properties and child collections of an entity are **not** rolled back in the event of a transaction rollback.

Caution: The service requires a connected database, and subsequent transactional methods can span the physical transaction. This means that it is possible to use the `TRANSACTION` option on a block and increase the transaction scope to span more than one service call. This, in principle, is an unsupported mode of operation. Attempting to use the `TRANSACTION` option around one of the few methods that require two transactions will prevent the second transaction from taking place (see Table updates that require two transactions on page 215), but there is no known reason why it should not work for other methods **as long as the transaction does not span multiple services**. If the services are connected to different databases, the operation would be a two-phase commit. However, the effect of a transaction that spans multiple instances on the same database is quite unpredictable.

The API is designed to run separately from the database in a future release, and code that has functionality that relies on the `TRANSACTION` option will likely not be able to run separate from the database in the future. Note that the API design is based on a common OpenEdge application strategy, and the warning against using the `TRANSACTION` option is consistent with OpenEdge recommendations for implementing a business logic layer in ABL; there are no current plans to support the API in a distributed application environment.

Table updates that require two transactions

The `UpdateTables( )` method might commit changes in two transactions if a table is enabled for multi-tenancy and also has partition changes. The partition changes are applied in a separate transaction because the partitions (`_storage-object` records) do not exist in the database until the transaction that enables the table for multi-tenancy is completed.

Note that the only way to update partition changes together with the change to enable the table for multi-tenancy is through the `Import( )` method on the `ISchema` object or on table collection objects. However, references to the `Partitions` collection property on these objects before the partitions exist in the database will throw an error.

Maintaining multi-tenant and table partition entities

The following sections describe common maintenance operations on the basic entities supported by the DataAdminService.

- Areas (creating and reading) on page 216
- Authentication systems (creating, reading, updating and deleting) on page 217
- Domains (creating, reading, updating and deleting) on page 218
- Schema (reading, loading, and updating) on page 219
- Sequences (creating, reading, updating and deleting) on page 220
- Tables (reading and updating) on page 220
- Table permissions (reading and updating) on page 221
- Tenants (creating, reading, updating and deleting) on page 222
- Tenant groups (creating, reading, updating and deleting) on page 223
- Users (creating, reading, updating and deleting) on page 224

- User permissions (reading) on page 225
- Partitions (creating, reading, updating, deleting, enabling and allocating) on page 225

Note: The code in these sections assume the necessary `USING` statements to define the interface and class packages.

Areas (creating and reading)

The following code shows how to use the API service to create and read an area using the `IArea` interface. The area cannot be deleted from the database after it has been created and consequently the service has no delete method:

```
define variable area as IArea no-undo.

area = service:NewArea("MyArea").
/* Set properties and add extents */
. . .
service:CreateArea(area).
```

Note: Creation is only supported on a local connection.

The properties cannot be edited after creation, but the update method is used to add extents to the area:

```
area = service:GetArea("MyArea").
/* Add extents */
. . .
service:UpdateArea(area).
```

It is also possible to retrieve a collection of areas in the database from the service and create areas and/or add extent to the areas in the `IAreaSet` collection before passing the collection to the service:

```
define variable areas as IAreaSet no-undo.
define variable area as IArea no-undo.

areas = service:Areas().

area = service:NewArea("OrderArea").
/* Set properties and add extents */
. . .
areas:Add(area).

area = areas:Find("InventoryArea").
/* Add extents */
. . .

service:UpdateArea(areas).
```

For more information on the:

- `IArea` entity interface, see IArea interface on page 710
- `IAreaSet` entity collection interface, see IAreaSet interface on page 775

Authentication systems (creating, reading, updating and deleting)

The following code shows how to use the API service to create, read, update, and delete single authentication systems using the `IAuthenticationSystem` interface:

```
define variable authSys as IAuthenticationSystem no-undo.

authSys = service:NewAuthenticationSystem("MyAuthSystem").
/* Set properties */
. . .
service:CreateAuthenticationSystem(authSys).

. . .
authSys = service:GetAuthenticationSystemAuthenticationSystem
  ("MyAuthSystem").
/* Set properties */
. . .
service:UpdateAuthenticationSystem(authSys).

. . .
service:DeleteAuthenticationSystem("MyAuthSystem ").
```

For more information on authentication systems, see *Introduction to Identity Management*.

It is also possible to retrieve a collection of authentication systems in the database from the service and then do create, read, update and/or delete on authentication systems in the `IAuthenticationSystemSet` collection before passing the collection to the service to be committed in a single transaction:

```
define variable typeSet as IAuthenticationSystemSet no-undo.
define variable authSys as IAuthenticationSystem no-undo.

typeSet = service:GetAuthenticationSystems().

authSys = service:NewAuthenticationSystem("SomeSystem").
/* Set properties and optionally add elements to collections */
. . .
typeSet:Add(authSys).

. . .
authSys = typeSet:Find("XSystem")
authSys:Description = ("XSystem is for special use").

authSys = typeSet:Find("ZSystem")
typeSet:Remove(authSys).

service:UpdateAuthenticationSystems(typeSet).
```

For more information on the:

- `IAuthenticationSystem` entity interface, see IAuthenticationSystem interface on page 712.
- `IAuthenticationSystemSet` entity collection interface, see IAuthenticationSystemSet interface on page 776.

Domains (creating, reading, updating and deleting)

The following code shows how to use the API service to create, read, update, and delete single domains (*security domains*) using the `IDomain` interface:

```
define variable domain as IDomain no-undo.

domain = service:NewDomain("MyDomain").
/* Set properties and optionally add elements to collections */
. . .
service:CreateDomain(domain).

. . .
domain = service:GetDomain("MyDomain").
/* Set properties */
service:UpdateDomain(domain).

. . .
service:DeleteDomain("MyDomain").
```

For more information on security domains, see *Introduction to Identity Management*.

It is also possible to retrieve a collection of domains in the database from the service and then do create, read, update and/or delete on domains in the `IDomainSet` collection before passing the collection to the service to be committed in a single transaction:

```
define variable domainSet as IDomainSet no-undo.

domainSet = service:GetDomains().

domain = service:NewDomain("SomeDomain").
/* Set properties and optionally add elements to collections */
. . .
domainSet:Add(domain).

. . .
domain = domainSet:Find("XDomain")
domain:IsEnabled = true.

domain = domainSet:Find("ZDomain")
domainMap:Remove(domain).

service:UpdateDomains(domains).
```

For more information on the:

- `IDomain` entity interface, see IDomain interface on page 718
- `IDomainSet` entity collection interface, see IDomainSet interface on page 780

Schema (reading, loading, and updating)

The `GetSchema( )` method reads the current schema from the database, returning it as an `ISchema` object. It has no key parameter because the API supports only the PUB-owned schema. (For more information on schema ownership, see *Manage the OpenEdge Database*.) Therefore, the returned `ISchema` object has collections of PUB-owned tables and sequences. This method is primarily supported for use by OpenEdge database tools. In your application code, you typically do not need to use this method because you can directly access all `ITable` and `ISequence` collections loaded from the database using the corresponding `Get*( )` and `Update*( )` service methods.

The `GetSchemaChanges( )` method returns an `ISchema` object containing data definitions loaded from a specified `.df` file. You can then set load options using the `LoadOptions` property on `ISchema` and pass the `ISchema` object to the `UpdateSchemaChanges( )` method to load the new or modified data definitions into the database. After loading the schema you can then access the schema partition collection using the `Partitions` property on `ISchema` to allocate or check the allocation status of the loaded schema partitions.

The following code shows how to use the API service to read schema definitions from a `.df` file and load them into the database using the resulting `ISchema` object:

```
define variable definitions as ISchema no-undo.

 definitions = service:GetSchemaChanges("myschema.df").
definitions:LoadOptions:ForceIndexDeactivation = true.
service:UpdateSchemaChanges(definitions).
/* Allocate new partitions */
. . .
service:UpdateSchema(definitions).
```

The service can only work on one instance of an `ISchema` retrieved with `GetSchemaChanges( )` until the instance has been loaded with `UpdateSchemaChanges( )`, which also creates new partitions for the loaded schema. An optional call to `UpdateSchema( )` loads any updates that you make to these partitions. The service also has a `CancelSchemaChanges( )` method that cancels the current schema changes and allows a new call to `GetSchemaChanges( )`.

The service also has a logical `LoadSchemaChanges( )` method that loads data definitions directly into the database from the data definitions (`.df`) file specified in its character parameter.

For more information on the `LoadOptions` and `Partitions` properties, see IPartitionOwner interface on page 729 and IPartitionCollection interface on page 785.

Sequences (creating, reading, updating and deleting)

The following code shows how to use the API service to create, read, update, and delete sequences using the `ISequence` interface:

```
define variable sequence as ISequence no-undo.

sequence = service:NewSequence("OrderSequence").
/* Set properties */
. . .
service:CreateSequence(sequence).

. . .
sequence = service:GetSequence("CustSequence").
/* Set properties */
. . .
service:UpdateSequence(sequence).

. . .
service:DeleteSequence("CustSequence").
```

It is also possible to retrieve a collection of sequences in the database from the service and then add, update, and remove sequences in the `ISequenceSet` collection before passing the collection to the service to commit all changes in a single transaction:

```
define variable sequenceSet as ISequenceSet no-undo.
define variable sequence as ISequence no-undo.
sequenceSet = service:GetUsers().
sequence = service:NewSequence("ItemSeq").
/* Set properties */
. . .
sequenceSet:Add(sequence).
sequence = sequenceSet:Find("OrderSequence").
sequence:Description = "Sequence for the Order table.".
sequence = sequenceSet:Find("ItemSeq").
sequenceSet:Remove(sequence).
service:UpdateSequences(sequenceSet).
```

For more information on the:

- `ISequence` entity interface, see ISequence interface on page 748
- `ISequenceSet` entity collection interface, see ISequenceSet interface on page 793

Tables (reading and updating)

The API provides access to tables for multi-tenant purposes using the `ITable` interface. The only writeable property is the `IsMultitenant` property, and then only when the table is not yet enabled for multi-tenancy.

The service returns tables that can be multi-tenant enabled and accessed with ABL code. The service does not expose the (hidden) schema tables in the Data Dictionary and tables that only can be accessed from SQL.

The following example shows how to enable a table for multi-tenancy:

```
define variable table as ITable no-undo.

table = service:GetTable("Customer").
table:IsMultitenant = true.
service:UpdateTable(table).
```

It is also possible to retrieve a collection of tables in the database from the service and then update the multi-tenant related properties in the instances on the `ITableSet` collection before passing the collection to the service to commit all changes in a single transaction.

For more information on the:

- `ITable` entity interface, see ITable interface on page 7513
- `ITableSet` entity collection interface, see ITableSet interface on page 796

Table permissions (reading and updating)

The API provides access to table permissions. The following code shows how to use the API service to read and update table permissions using the `ITablePermission` interface:

```
define variable tablePerm as ITablePermission no-undo.

table = service:GetTablePermission("Customer").
/* Set properties */
. . .
service:UpdateTablePermission(tablePerm).
```

It is also possible to retrieve a collection of tables in the database from the service and then update the `Can*` properties and multi-tenant related properties in the instances on the `ITablePermissionSet` collection before passing the collection to the service to commit all changes in a single transaction.

For more information on the:

- `ITablePermission` entity interface, see ITablePermission interface on page 754
- `ITablePermissionSet` entity collection interface, see ITablePermissionSet interface on page 795

Tenants (creating, reading, updating and deleting)

The following code shows how to use the API service to create, read, update, and delete tenants using the `ITenant` interface:

```
define variable tenant  as ITenant no-undo.

tenant = service:NewTenant("Andersen").
/* Set properties and optionally add elements to collections */
. . .
service:CreateTenant(tenant).

. . .
tenant = service:GetTenant("Acme").
/* Set properties and/or add/delete domain(s) */
. . .
service:UpdateTenant(tenant).

. . .
service:DeleteTenant("Andersen").
```

It is also possible to retrieve a collection of all the tenants in the database from the service and then do create, read, update, and delete on tenants in the `ITenantSet` collection before passing the collection to the service to be committed in a single transaction:

```
define variable tenantSet as ITenantSet no-undo.
define variable tenant  as ITenant no-undo.

tenantSet = service:GetTenants().
tenant = service:newTenant("Andersen").
/* Set properties and optionally add elements to collections */
. . .
tenantSet:Add(tenant).

tenant = tenantSet:Find("Johnson")
tenant:Description = "Johnson is a paying customer.".

tenant = tenantSet:Find("Smith").
tenantSet:Remove(tenant).

service:UpdateTenants(tenantSet).
```

Getting a tenant by an external ID

The service also has a `GetTenantByExternalId( CHARACTER )` method that allows you to retrieve a tenant by a user-defined external id.

Note: The `ExternalId` property is not defined as a unique key and `GetTenantByExternalId( )` will throw an error if more than one tenant is found with the requested external ID. One can use `GetTenants( )` with a filter to retrieve a collection of tenants with a specific external ID.

For more information on the:

- `ITenant` entity interface, see ITenant interface on page 755
- `ITenantSet` entity collection interface, see ITenantSet interface on page 798

Tenant groups (creating, reading, updating and deleting)

The following code shows how to use the API service to create, read, update, and delete tenant groups using the `ITenantGroup` interface:

```
define variable group as ITenantGroup no-undo.

group = service:NewTenantGroup("CustGroup").
/* Set properties and optionally add elements to collections */
. . .
service:CreateTenantGroup(group).

. . .
group = service:GetTenantGroup ("ACMECustGroup").
/* Set properties and/or add/delete domain(s) */
. . .
service:UpdateTenantGroup(group).
. . .

service:DeleteTenantGroup("CustGroup").
```

It is also possible to retrieve a collection of tenant groups in the database from the service and then do create, read, update, and delete on tenants in the `ITenantGroupSet` collection before passing the collection to the service to be committed in a single transaction:

```
define variable groupSet as ITenantGroupSet no-undo.
define variable group    as ITenantGroup no-undo.

groupSet = service:GetTenants().
group = service:NewTenantGroup("ABCGroup").
/* Set properties and optionally add elements to collections */
. . .
groupSet:Add(group).
group = groupSet:Find("ABCCustomer")
group:Description = "ABCCustomer group.".
group = groupSet:Find("ABCOrder").
groupSet:Remove(group).
service:UpdateTenantGroups(groupSet).
```

For more information on the:

- `ITenantGroup` entity interface, see ITenantGroup interface on page 761
- `ITenantGroupSet` entity collection interface, see ITenantGroupSet interface on page 797

Users (creating, reading, updating and deleting)

The following code shows how to use the API service to create, read, update, and delete single users using the `IUser` interface:

```
define variable usr as IUser no-undo.

usr = service:NewUser("MyUser@MyDomain").
/* Set properties */
. . .
service:CreateUser(usr).

. . .
usr = service:GetUser("MyUser@MyDomain").
/* Set properties */
. . .
service:UpdateUser(usr).

. . .
service:DeleteUser("MyUser@MyDomain").
```

It is also possible to retrieve a collection of users in the database from the service and then do create, read, update, and delete on users in the `IUserSet` collection before passing the collection to the service to be committed in a single transaction:

```
define variable userSet as IUserSet no-undo.

userSet = service:GetUsers().

usr = service:NewUser("Fred@ABCDomain").
/* Set properties */
. . .
userSet:Add(usr).

usr = userSet:Find("Admin").
usr:Description = "The Admin user for the default domain.".

usr = userSet:Find("Fred ").
userSet:Remove(usr).

service:UpdateUsers(userSet).
```

Users can also be created, read, updated, and deleted in a similar manner on the `Users` property collection available on the `IDomain` interface. When creating a new `IUser` to add to the `IDomain Users` collection the user name should be passed to the constructor **without** the *@domain* qualifier, as follows:

```
domain = service:GetDomain("ZZDomain"):
users = domain:Users().

user = service:NewUser("John").
/* Set properties */
. . .
users:Add(user).

service:UpdateDomain(domain).
```

For more information on the:

- `IUser` entity interface, see IUser interface on page 766
- `IUserSet` entity collection interface, see IUserSet interface on page 801

User permissions (reading)

The following code show how to use the API service to read permissions for a user using the `IUserPermisson` interface:

```
define variable userPerm as IUserPermission no-undo.

 userPerm = service:GetUserPermission("MyUser").
```

It is also possible to retrieve a collection of user permissions in the database from the service and then update the user permissions in the `IUserPermissionSet` collection before passing the collection to the service to be committed in a single transaction:

```
define variable userPermSet as IUserPermissionSet no-undo.

userPermSet = service:GetUserPermissions().
```

For more information on the:

- `IUserPermisson` entity interface, see IUserPermission interface on page 769
- `IUserPermissonSet` entity collection interface, see IUserPermissionSet interface on page 800

Partitions (creating, reading, updating, deleting, enabling and allocating)

The following code shows how to use the API service to create, read, update, and delete partition defaults, enable and allocate partitions using the IPartitionParent interface. The IPartitionParent interface abstracts common properties for all interfaces and classes that define partition defaults:

```
define variable partition as IPartitionParent no-undo.

partition = service:NewPartition("Andersen").
/* Set properties and optionally add elements to collections */
. . .
service:CreatePartition(partition).

. . .
partition = service:GetPartition("Acme").
/* Set properties and/or add/delete domain(s) */
. . .
service:UpdatePartition(partition).

. . .
service:DeletePartition("Andersen").
```

There are currently no service methods or utilities that reference this interface. The new IPartitionPolicy and IPartitionOwner interfaces inherit this interface.

For more information on the:

- `IPartitionParent` entity interface, see IPartitionMap interface on page 787
- `IPartitionOwner` entity interface, see IPartitionPolicy interface on page 733
- `IPartitionPolicy` entity interface, see IPartitionPolicy interface on page 733
- `IPartitionPolicyDetail` entity interface, see IPartitionPolicyDetail interface on page 739
- `IPartitionPolicySet` entity collection interface, see IPartitionPolicySet interface on page 790
- `IPartitionPolicyDetailSet` entity collection interface, see IPartitionPolicyDetailSet interface on page 789

Working with partitions

The IPartitionPolicyDetail partition collection property exposes partitions for the table partition detail. The property returns IPartitionMap. This is mapped using the ISchemaElement (table, index or field) as key.

The following code illustrates how to allocate partitions of an existing policy to different areas:

```
block level on error undo, throw.
using OpenEdge.DataAdmin.*.
using OpenEdge.DataAdmin.Lang.Collections.*.

define variable service as DataAdminService no-undo.

define variable tableImpl as ITable no-undo.
define variable iterator as IIterator no-undo.
define variable iterator2 as IIterator no-undo.
define variable partition as IPartition no-undo.
define variable policy as IPartitionPolicy no-undo.
define variable detail as IPartitionPolicyDetail no-undo.
define variable areaNameSuffix as character no-undo.

assign
    service = new DataAdminService("Sports2000")
    policy = service:GetPartitionPolicy("EmployeePartition").
iterator = policy:Details:Iterator().
do while iterator:HasNext():
    detail = cast(iterator:Next(),IPartitionPolicyDetail).

   /** This is assuming that you have created 3 areas for each detail */
areaNameSuffix = "-" + detail:name.
iterator2 = detail:Partitions:Iterator().
    do while iterator2:HasNext():
  partition = cast(iterator2:Next(),IPartition).
        case partition:ObjectType:
            when "Table" then
            do:
              partition:Area = service:GetArea("DataArea" + areaNameSuffix).
            end.
            when "Index" then
            do:
              partition:Area = service:GetArea("IndexArea" + areaNameSuffix).
end.
            when "Field" then
            do:
partition:Area = service:GetArea("LobArea" + areaNameSuffix).
            end.
        end case.
    end.
detail:Allocate().
end.

service:UpdatePartitionPolicy(policy).
```

Splitting a Partition

You split a partition when you move table data from one partition to another. You must use the `IParitionSplitUtility` utility to split a partition. For more information on splitting a partition, see PartitionMergeUtility class on page 233.

Filtering and sorting

The API provides overloaded filter versions of all `Get*( )` methods in the DataAdminService. The collection `Get*( )` methods provide overloadings with a filter character parameter to control the amount of data returned in the collection. In addition, all `Get*( )` methods have overloadings that expect an instance of the `IRequestInfo` interface that can provide control of all child collections of objects.

The common `Iterator( )` method is also overloaded to support optional sort and filter parameters.

Filter expressions

The API expects the filter to be expressed as a character expression, passed directly to the collection `Get*( )` methods, or assigned to properties of `IRequestInfo` instances (see Filtering on a child collection using the RequestInfo class on page 228).

The query expression syntax is similar to any ABL expression with the following differences:

- The expression references properties of the class.
- The values in the property expression must be quoted irrespective of data type. This is similar to the requirement for dynamic queries to handle numeric and date formats according to session settings, but in this case it is not optional, and is required for all data types, including `LOGICAL`.
- Child collections that represent many to many relations support a qualifier to query the object at the other end of the relation.

The following code shows the filter used to retrieve tenants that have not yet been allocated in order to allocate them:

```
define variable tenantSet as ITenantSet no-undo.
define variable iterator as  IIterator no-undo.

tenantSet = service:GetTenants("IsAllocated = 'false'").
iterator = tenantSet:Iterator().
do while iterator:HasNext():
    cast(iterator:Next(),ITenant):Allocate().
end.
```

The following code shows the filter used to retrieve Type 2 areas. This is an example of a query that cannot be directly expressed against the area table in the database:

```
define variable areaSet as IAreaSet no-undo.
areaSet = service:GetAreas("IsType2 = 'true'").
```

Filtering on a child collection using the RequestInfo class

To filter requested objects using the `RequestInfo` class, you instantiate the `RequestInfo` class using filter parameters and access the instance as the `IRequestInfo` interface type. Constructors for the `RequestInfo` class allow you to specify filters on child collections of requested objects for all `Get*( )` methods that return an object that has child collections. The top instance represents the requested object and the child instances are added using an `Add( )` method for the collection that is to be filtered.

You must specify a unique key name and value for an `IRequestInfo` instance that you use with `Get*( )` methods that return a single `IDataAdminElement`. To specify criteria for a collection, you specify the collection name using the single parameter constructor and assign the query to the `QueryString` property.

The following code example shows the retrieval of a single tenant with a filter on disabled domains:

```
define variable requestInfo as IRequestInfo no-undo.
define variable childInfo as IRequestInfo no-undo.
define variable tnt as ITenant no-undo.

requestInfo = new RequestInfo("Name","ABXTenant").
childInfo = new RequestInfo("Domains").
childInfo:QueryString = ("IsEnabled = 'false'").
requestInfo:Add(childInfo).
tnt = service:GetTenant(requestInfo).
```

The following code example shows how to use of the `IRequestInfo` object to retrieve a collection of non-allocated tenants with disabled domains (similar to the previous example). In this case, all the retrieved tenants will only have domains that satisfy the filter criteria. Note that you need an `IRequestInfo` object for the top level even if you don't want to filter the top collection, as with the tenants this case. `IRequestInfo` children are only added for the child collections that need a filter:

```
define variable requestInfo as IRequestInfo no-undo.
define variable childInfo as IRequestInfo no-undo.
define variable tntSet as ITenantSet no-undo.

requestInfo = new RequestInfo("Tenants").
requestInfo:QueryString = ("IsAllocated = 'false'").
childInfo = new RequestInfo("Domains").
childInfo:QueryString = ("IsEnabled = 'false'").
requestInfo:Add(childInfo).
tntSet = service:GetTenants(requestInfo).
```

The `IRequestInfo` interface defines several public properties and methods that you can use to manage entity and collection sorting and filtering.

IRequestInfo interface properties

The following table lists the properties defined by the interface.

Table 27: Multi-tenant Maintenance IRequestInfo interface properties

Property name	Data type	Access	Description
`KeyFields`	`CHARACTER`	Read-only	Returns a comma-separated list of the key field names used to instantiate `RequestInfo`. For example, given the following statement: `requestInfo = new RequestInfo ("Name","ABXTenant")`. The `requestInfo:KeyFields` property contains `"Name"`.
`Name`	`CHARACTER`	Read-only	Returns the entity name of a collection passed to the `RequestInfo` constructor. For example, given the following statement: `childInfo = new RequestInfo("Domains")`. The `childInfo:Name` property will contains `"Domains"`.
`PageSize`	`INTEGER`	Readable, Writeable	Reserved by OpenEdge.
`QueryString`	`CHARACTER`	Readable, Writeable	A query string on the properties of the specified entity collection, specified according to rules for filter expressions. For more information, see Filter expressions on page 227.
`SearchString`	`CHARACTER`	Readable, Writeable	Reserved by OpenEdge.

The most common property for use in application code is `QueryString`. The `SearchString` property is reserved for future use, and the `PageSize` property is for use by OpenEdge database tools. Setting `PageSize` in a collection for iteration or use in application code is not supported. It might limit the number of items in the collection as long as there are no other requests to the same collection, but there is no way to ask for a next batch.

IRequestInfo interface methods

The following table lists the public methods defined by the interface.

Table 28: Multi-tenant Maintenance IRequestInfo interface methods

Method syntax	Description
`LOGICAL Add ( child AS IRequestInfo )`	Adds a collection request specified by *child* with its Name property set to match a collection.
`LOGICAL Add ( name AS CHARACTER , child AS IRequestInfo )`	Adds a collection request specified by *child* with its *name* parameter set to match a collection.
`IRequestInfo Get ( name AS CHARACTER )`	Gets a child collection request specified by *name*.
`IRequestInfo EXTENT GetChildren ( )`	Returns an array of all child collection requests.
`CHARACTER EXTENT GetKeyValues ( )`	Gets the key values for the top-level collection request.
`LOGICAL Remove ( child AS IRequestInfo )`	Removes the collection request specified by *child* with its Name property set to match a collection.

Exporting and importing

All interfaces, for both entities and collections, support the import and export of data values between objects and JavaScript Object Notation (JSON) files. For information on working with JSON using ABL, see *Use JSON with ABL Applications*. You can also export tenants and tenant groups as ABL code generated to procedure files.

Exporting and importing objects as JSON files

Both entity and entity collection objects support overloadings of the `Export( )` and `Import( )` methods using a format to store and retrieve object data similar to the ABL `WRITE-JSON( )` and `READ-JSON( )` methods (respectively) on a temp-table. In addition, these objects support `ExportTree( )` and `ImportTree( )` methods that export and import an entire object, including all child collections. The `ExportTree( )` method is overloaded with a parameter to specify what collections to export. The `ExportTree( )` and `ImportTree( )` methods use a format to store and retrieve object data similar to the ABL `WRITE-JSON( )` and `READ-JSON( )` methods (respectively) on a ProDataSet.

The following code exports all tenants in the database into a JSON (`.json`) file:

```
tenantCol  = service:GetTenants().
tenantCol:Export("mytenants.json").
```

The following code loads new tenants from a JSON file and into the database:

```
tenantCol  = new TenantSet().
tenantCol:Import("mytenants.json").
service:CreateTenants(tenantCol).
```

This import can also be done on existing tenants as shown using the `ImportTree( )` method in a following example.

The following code exports all tenants and partitions in the database into a JSON file:

```
tenantCol  = service:GetTenants().
tenantCol:ExportTree("mytenants.json","partitions").
```

The following code imports tenants and any child objects from the specified JSON file into the database:

```
tenantCol  = service:GetTenants().
tenantCol:ImportTree("mytenants.json").
service:UpdateTenants(tenantCol).
```

This import can also be done for new tenants as shown using the `Import( )` method in a previous example.

Exporting tenants and tenant groups as ABL code

Tenants and tenant groups can be exported as ABL code that can be used to load the tenants and tenant groups into a different database.

The following code generates code to create a tenant domains, tenant groups and partitions in a database. Note that the reference to domains will also generate code to create users of the domains:

```
tenant = service:GetTenant("MyTenant").
tenant:ExportTree("mytenant.p","domains,tenantgroups,partitions").
```

The following code generates code to create a tenant group with partitions in a database:

```
tenantGroup = service:GetTenantGroup("MyTenantGroup").
tenantGroup:ExportTree("mytenant.p","partitions").
```

You can generate code that also adds tenants to a tenant group as follows. In this case the code that generates the corresponding tenant should not include the tenant groups:

```
tenantGroup = service:GetTenantGroup("MyTenantGroup").
tenantGroup:ExportTree("mytenant.p","tenants,partitions").
```

Exporting and Importing Partition Policies and Details

Partition Policies and Partition Policy Details can be dumped and loaded in standard ABL .d format using standard Data Admin Utilities. Note that the utility only know these by their physical table name _partition-policy and _partition-policy-detail and display this in the user interface and expect this as input in the dump and load .p APIs. This also includes OEM table dump and load support. The DataAdmin Utilities allows both the class name and physical database name to be used.

The data that are written to the .d does NOT correspond to the physical storage. These tables have relations that are stored with internal identifiers that cannot be used to export for the purpose of importing (recreating) the tables in another database instance.

The standard .d backwards compatibility rule/contract applies in the sense that the order of the data in the .d will not change in future releases and new fields/properties will always be added to the end of the exiting list to ensure that a .d can be safely imported in a new version.

The export and import methods of the collections for these classes also support the .d format in addition to the JSON format. The type is specified and identified by the extension of the file name passed in the file name parameter.

WARNING: These methods do not support or handle code page and format information in the .d file and will use current session settings both for export and import. They should thus only be used for short lived export/import in the same session or in cases where this is controlled. It is important to use the Data Admin table dump and load tools for regular data management purposes to ensure that the data includes code page and format settings that allows data to be safely loaded independent of session settings.

One benefit of the export and import methods versus data admin table utilities is that the collections can be filtered allowing export of subsets of data.

Executing DataAdminService utilities

The DataAdminService has an `ExecuteUtility( )` service method, which executes supported utility operations. These can, in principle, be any type of operation that is not a regular CRUD operation supported by the entity service methods. The main purpose is to support existing data administration utilities (like dump and load) and OpenEdge command-line utilities like PROUTIL.

The supported utility operations are provided by a single `IDataAdminUtility` interface and a set of implementing utility classes. The utility interface and classes are in the `OpenEdge.DataAdmin.Util` package. The current release supports `DeallocateUtility, PartitionMergeUtility,` and `PartitionSplitUtility` class.

Note: The utility operations specified for the `ExecuteUtility( )` service method are only supported on a local connection.

IDataAdminUtility interface

`OpenEdge.DataAdmin.Util.IDataAdminUtility` is the common interface for all DataAdminService utilities. This interface does not support any public properties or methods for use by application code, but allows you to pass all supported utility classes to the `ExecuteUtility( )` service method.

DeallocateUtility class

The `DeallocateUtility` class allows you to deallocate entity partitions. This is a wrapper around the PROUTIL command. It is implemented as a utility because it is likely to be commonly used when working with multi-tenant databases.

The class supports two constructors, which allows you to deallocate a table for a tenant and a tenant group, respectively:

Syntax

```
DeallocateUtility( ITenant, ITable )
DeallocateUtility( ITenantGroup )
```

The following code shows how to deallocate a table for a tenant:

```
define variable util as IDataAdminUtility no-undo.
define variable tbl as ITable no-undo.
define variable tnt as ITenant no-undo.
tbl = service:GetTable("Customer").
tnt = service:GetTenant("ABC").
util = new DeallocateUtility(tnt,tbl).
service:ExecuteUtility(util).
```

The following code shows how to deallocate a tenant group:

```
define variable util as IDataAdminUtility no-undo.
define variable grp as ITenantGroup no-undo.
grp = service:GetTenantGroup("Customer").
util = new DeallocateUtility(grp).
service:ExecuteUtility(util).
```

PartitionMergeUtility class

Using the PartitionMergeUtility class, you can merge data from one or more partitions into a single target partition. Merging partitions is beneficial for simplifying a partitioned environment and for placing historical data into a single partition. When you merge partitions, the range of the target partition is set to the combined range of the merged partitions. The range value in the partition policy detail of the target partition is set to the highest range value of the merged partitions.

Syntax

```
PartitionMergeUtility(start-partition, [ end-partition, ] target-partition)
```

You can use this utility in the following scenarios:

- Merge one partition into a target partition on page 234
- Merge multiple partitions into a target partition on page 234

Merge one partition into a target partition

To merge one partition into a target partition, specify the `start-partition` and `target-partition` parameters in the PartitionMergeUtility class. In the merge operation, the data from the start partition is moved into the target partition and then the start partition is deleted.

The following sample code shows how to merge one partition into a target partition:

```
block-level on error undo, throw

using Progress.Lang.Error from propath.
using OpenEdge.DataAdmin.DataAdminService from propath.
using OpenEdge.DataAdmin.Error.DataAdmin.ErrorHandler from propath.
using OpenEdge.DataAdmin.IPartitionPolicy from propath.
using OpenEdge.DataAdmin.IPartitionPolicyDetail from propath.
using OpenEdge.DataAdmin.IPartition from propath.
using OpenEdge.DataAdmin.ITable from propath.
using OpenEdge.DataAdmin.Util.PartitionMergeUtility from propath.

define variable service as DataAdminService no-undo.
define variable errorHandler as DataAdminErrorHandler no-undo.
define variable tbl as ITable no-undo.
define variable merge as PartitionMergeUtility no-undo.
define variable policy as IPartitionPolicy no-undo.
define variable start as IPartitionPolicyDetail no-undo.
define variable target as IPartitionPolicyDetail no-undo.

/* Start a service for the sptp database */
service = new DataAdminService("spmerge").

/* Retrieve the table */
tbl = service:GetTable("order").

/* Retrieve the partition policy */
policy = service:GetPartitionPolicy("pol")

start = policy:Details:Find("pol1").
target = policy:Details:Find("pol2").

/* Pass the start and target partitions to the utility for merging the start
partition into the target partition */
merge = new PartitionMergeUtility(start, target).
service:ExecuteUtility(merge).

catch e as Progress.Lang.Error:
    errorHandler = new DataAdminErrorHandler().
    errorHandler:Error(e).
end catch.

finally:
    delete object service no-error.
end finally.
```

Merge multiple partitions into a target partition

To merge multiple partitions into a target partition, specify the `start-partition`, `end-partition`, and `target-partition` parameters in the PartitionMergeUtility class. If the start and end partitions are not adjacent to each other, in addition to the start and end partitions, the data from all the partitions between the two is merged into the target partition. Finally, all the partitions that have been merged into the target partition are deleted.

PartitionSplitUtility class

You split a partition when you move table data from one partition to another.

Note: `PartitionSplitUtility` can be used only on a range or list-range type partition policy.

Syntax

```
PartitionSplitUtility( ITable,[ IIndex,] Transaction-size)
```

You can use this utility in the following scenarios:

- Split from a composite partition on page 235
- Split from an allocated partition detail on page 237

Split from a composite partition

The table data of a partitioned table resides in the composite partition (the default partition) until the table data is moved to the respective partitions as per the table partition policy. You can mark partition details as split targets and then split them incrementally from the composite partition to dedicated partitions.

The following sample code shows how to split from a composite partition:

```
block-level on error undo, throw

using OpenEdge.DataAdmin.Lang.Collections.IIterator from propath.
using OpenEdge.DataAdmin.DataAdminService from propath.
using OpenEdge.DataAdmin.IPartitionPolicy from propath.
using OpenEdge.DataAdmin.IPartitionPolicyDetail from propath.
using OpenEdge.DataAdmin.Util.PartitionSplitUtility from propath.
using OpenEdge.DataAdmin.Error.DataAdminErrorHandler from propath.

define variable service as DataAdminService no-undo.
define variable policy as IPartitionPolicy no-undo.
define variable detail as IPartitionPolicyDetail no-undo.
define variable iter as IIterator no-undo.
define variable split as PartitionSplitUtility no-undo.
define variable errorhandler as DataAdminErrorHandler no-undo.

assign
    service = new DataAdminService(ldbname(1))
    /* get the policy */
    policy = service:GetPartitionPolicy("OrderPolicy")

    /* get an iterator of policy details that are still in composite*/
    iter = policy:Details:Iterator().

do while iter:HasNext():
    detail = cast(iter:Next(), IPartitionPolicyDetail).
detail:IsSplitTarget = true.
end.

/* commit the splittarget changes */
service:UpdatePartitionPolicy(policy).

/* Pass the policy and (optional) transaction size to the constructor to split
the details from the composite partition */
split = new PartitionSplitUtility(policy, 100).
service:ExecuteUtility(split).

catch e as Progress.Lang.Error:
    errorHandler = new DataAdminErrorHandler().
    errorHandler:Error(e).
end catch.

finally:
    delete object service no-error.
end finally.
```

Partition details are marked as split targets. Using `PartionSplitUtility`, data from the composite partition is split to the marked partition details.

When splitting a composite partition, you can alternatively pass the table, `ITable`, to `PartionSplitUtility`.

The following sample code-snippet illustrates how to split a partition using the `ITable` interface:

```
using OpenEdge.DataAdmin.ITable from propath.
using OpenEdge.DataAdmin.DataAdminService from propath.
using OpenEdge.DataAdmin.IDataAdminService from propath.
using OpenEdge.DataAdmin.Util.PartitionSplitUtility from propath.

define variable myTable as ITable no-undo.
define variable service as IDataAdminService no-undo.
define variable split as PartitionSplitUtility no-undo.

assign
    service = new DataAdminService(ldbname(1))
    myTable = service:GetTable("myPolicy")

/* Pass the table to the constructor to split the composite partition, set transaction
 size to 100  */
split = new PartitionSplitUtility(myTable, 100).
service:ExecuteUtility(split).
```

Split from an allocated partition detail

A partition detail that has already moved to an allocated partition can be re-allocated to other partitions. That is, you can move data from an allocated partition to one or more new partitions.

The following sample code shows how to split from an allocated partition:

```
using OpenEdge.DataAdmin.DataAdminService from propath.
using OpenEdge.DataAdmin.Error.DataAdminErrorHandler from propath.
using OpenEdge.DataAdmin.ITable from propath.
using OpenEdge.DataAdmin.Util.PartitionSplitUtility from propath.
using OpenEdge.DataAdmin.IPartitionPolicy from propath.
using OpenEdge.DataAdmin.IPartitionPolicyDetail from propath.

define variable service as DataAdminService no-undo.
define variable policy as IPartitionPolicy no-undo.
define variable detail as IPartitionPolicyDetail no-undo.
define variable errorhandler as DataAdminErrorHandler no-undo.
define variable split as PartitionSplitUtility no-undo.

/* Retrieve the partition policy */
assign
    service = new DataAdminService(LDBNAME(1))
    policy = service:GetPartitionPolicy("RPolicy").

/* add a new detail and mark as splitTarget */
detail = service:NewPartitionPolicyDetail("MyPolicy-A").
policy:Details:Add(detail).
detail:SetValue("A").
detail:IsSplitTarget=true.

/* add a new detail and mark as splitTarget */
detail = service:NewPartitionPolicyDetail("MyPolicy-B").
policy:Details:Add(detail).
detail:SetValue("B").
detail:IsSplitTarget=true.

/* commit the new details */
service:UpdatePartitionPolicy(policy).
/* Run PartitionSplitUtility to split the detail that currently has the data
for the new details (all values with A, B and C)  */
detail = policy:Details:Find("MyPolicy-C").
split = new PartitionSplitUtility(detail).
service:ExecuteUtility(split).

catch e as Progress.Lang.Error:
    errorHandler = new DataAdminErrorHandler().
    errorHandler:Error(e).
end catch.

finally:
    delete object service no-error.
end finally.
```

Two new partition details, `MyPolicy-A and MyPolicy-B,` are created. Using `PartionSplitUtility`, data from the allocated partition detail, `MyPolicy-C`, is moved to the new partition details.

Handling errors

The API has a set of error classes that inherit the built-in `Progress.Lang.AppError` class. These classes encapsulate the most common error conditions. All errors are thrown back to your application, which can catch or throw them. You do not typically need to know what type of error is displayed or reference the error in your application code. If your application does not catch the error or use the `ROUTINE-LEVEL, ON ERROR UNDO, THROW` statement, the error messages display like any OpenEdge error message.

The API also provides some error handling mechanisms that you can also add to your application that work specifically with data errors returned by the DataAdminService.

Handling data errors with the DataAdminErrorHandler class

You can use the `OpenEdge.DataAdmin.Error.DataAdminErrorHandler` class to display data errors in code that uses the API. The default behavior is to display all data errors and the stack trace in a message box.

Data errors occur, for example, when a data commit fails validation or from some other data access problem. By default, these errors are not thrown from the service object that generates them because any attempted data updates might still be possible and the issue that caused the failure corrected. However, this error correction is difficult if the error is thrown.

In the following code, added in any block where the API is called, the `Error( )` method on `DataAdminErrorHandler` displays the error messages thrown from the `DataAminService` class and its related classes:

```
using Progress.Lang.*.
using OpenEdge.DataAdmin.Error.*.

catch e as Error :
    errorHandler =  new DataAdminErrorHandler().
    errorHandler:Error(e).
end catch.
```

Controlling data errors with the IDataAdminService interface

The DataAdminService provides a set of properties and methods that allow you to manage the data errors raised by the service. These are available through the `IDataAdminService` interface, like other service methods.

IDataAminService interface public properties for error handling

You can use the properties in the following table to manage data errors raised by the service.

Table 29: Multi-tenant Maintenance IDataAminService interface class error properties

Property name	Data type	Access	Description
`ThrowDataErrors`	`LOGICAL`	Readable, Writeable	Set to FALSE to specify that data errors are **not** to be thrown by the service. This ensures that all `Create*( )` and `Update*( )` methods return FALSE when a data error occurs instead of throwing an error.[8]

IDataAminService interface public methods for error handling

When you set `ThrowDataErrors` to FALSE, each time a `Create*( )` or `Update*( )` service method returns FALSE, you can use the methods in the following table to access and manage the data error that was raised by the service.

[8] The Create*( ) and Update*( ) methods might continue to throw errors that are not considered to be data errors, such as errors that result from illegal arguments, unsupported operations, invalid objects and handles, and so on.

Table 30: Multi-tenant Maintenance IDataAminService interface error methods

Method syntax	Description
`Progress.Lang.Error RemoveError ( )`	Retrieves and removes the current data error raised in the DataAdminService so you can, throw, display, or process the error in any way your application requires.
`VOID ThrowError ( )`	Throws the current data error raised in the DataAdminService.

5

Auditing

Auditing is a means of securely tracking and recording a trail of events that occur during execution of an application. The basic security in auditing allows this trail of events (audit events) to be recorded in a manner that cannot be changed or otherwise repudiated after the fact. OpenEdge provides core support for recording secure audit trails in any OpenEdge RDBMS that is enabled for auditing.

Auditing occurs for any OpenEdge application that connects to a database that is enabled for auditing, based on its auditing configuration. For some auditing events, OpenEdge automatically records them in the audit trail, based on active audit policies (rules) that govern how audit trails are defined, tracked, and recorded. For other auditing events, you must instruct the application to record them using ABL (Advanced Business Language) elements, and open these events, again, based on active audit policies.

You can also develop custom tools in ABL for auditing configuration and management. OpenEdge supports additional ABL elements and procedure APIs for this purpose.

For more information on auditing and how to configure and enable auditing in an OpenEdge RDBMS, see *Introduction to Security and Auditing*.

For details, see the following topics:

- Setting up application security for auditing
- Recording application events
- Developing auditing tools

Setting up application security for auditing

To provide auditing in an application, you generally need to consider three primary security requirements:

- Establish the auditing identity (ID).

- Verify that auditing is enabled for at least one connected database.
- Ensure that auditing privileges have been set appropriately to allow the auditing operations required.

The *auditing identity* (auditing ID) represents the user identity that OpenEdge associates with all audit events that it records in an audit-enabled database. Together with audit data integrity features, the auditing ID helps to ensure that the contents of the audit trail cannot be repudiated, either through unauthorized modification of the audit data or denial that a given user identity was involved with generating the audit data. Whatever user identity is the auditing ID for a given audit-enabled database becomes associated with all audit events, whether generated automatically by OpenEdge or explicitly by your application.

OpenEdge provides the option to establish the auditing ID for a database from one of the following sources:

- Database connection identity (ID)
- OpenEdge session identity (ID)

Because all OpenEdge auditing is configured and stored in a connected OpenEdge RDBMS, you must consider a combination of database configuration and application coding options to establish the effective auditing ID for an application.

The following sections describe:

- Configuring the auditing identity on page 242
- Asserting the auditing identity on page 243
- Verifying that auditing is enabled on page 245
- Configuring auditing privileges on page 245

Configuring the auditing identity

By default, OpenEdge uses the database connection ID (regardless of how you set it) as the auditing ID for a given audit-enabled database. You can set the database connection ID from a user ID that you authenticate to one of the following authentication systems, depending on the mechanism you use to set it:

- The OpenEdge internal authentication system (`_User` table)
- An external authentication system validated using a database or application trusted domain registry

For more information authentication systems and trusted domain registries, see the information on identity management in *Introduction to Security and Auditing*.

Note: Typically, both to better secure your database and to ensure that there is a meaningful auditing ID, you can select the **Disallow Blank UserId** option, also in the **Database Options** dialog box. This prevents a blank user ID from connecting to the database.

Instead of using the database connection ID, you can specify the OpenEdge session ID as the source for the auditing ID (regardless of the database connection ID setting), using the Data Administration tool or the character-mode Data Dictionary.

To specify the OpenEdge session ID as the source for the auditing ID, in the **Database Options** dialog box (accessed from the **Admin** menu), select the **Use Application User Id for Auditing** option.

When you select this option, OpenEdge uses the OpenEdge session ID (regardless of how it is set) as the auditing ID for the configured database. For more information on setting the auditing ID in your application code, see Asserting the auditing identity on page 243. You typically set the OpenEdge session ID from a user ID that you authenticate to an external authentication system and validate to a corresponding authentication domain entry in the application trusted domain registry.

You can also achieve the same effect of having the auditing ID set from the OpenEdge session ID, by setting another database option, **Trust Application Domain Registry**. With this option set, when you assert an externally authenticated user ID as the OpenEdge session ID this also asserts the database connection ID using the same user ID, which by default is the auditing ID.

For more information on the database connection ID and the OpenEdge session ID, see Application Security on page 97.

The following section provides more information on how you can set the auditing ID in your application, depending on the user identity you use as its source.

Asserting the auditing identity

The ABL mechanisms for asserting the auditing ID depend on how you authenticate and assert user IDs. These mechanisms include several ABL elements, including:

- The ABL `SETUSERID` function, `CONNECT` statement, and the User ID (-U) and Password (-P) startup parameters used to authenticate and set database connection IDs using the OpenEdge internal authentication system (`_User table`)
- The client-principal object used to represent and, together with a trusted authentication domain registry, validate an externally authenticated user ID as a database connection ID or OpenEdge session ID
- The ABL `SET-DB-CLIENT` function and the `SET-CLIENT( )` method on the `SECURITY-POLICY` system handle used to assert and validate an externally authenticated user ID represented by a client-principal object

The following sections describe the use of these elements in the context of auditing. For more information on using these mechanisms to assert user identities in application code, see Application Security on page 97.

This section describes the following basic application scenarios and the ABL elements you can code to assert the auditing ID for a given audit-enabled database:

- Using the database connection ID on page 243
- Using the OpenEdge session ID on page 244
- Application design considerations on page 244

Using the database connection ID

When you configure auditing to use the database connection ID (the default), you enable each audit-enabled database to use a different auditing ID that is potentially authenticated to a different authentication system. You can use all of the following ABL elements to assert the database connection ID as the auditing ID:

1. User ID (`-U`) and Password (`-P`) startup and connection parameters (at client startup or in the `CONNECT` statement), which authenticate and assert a connection ID from the database `_User` table
2. `SETUSERID` function, which authenticates and asserts a connection ID to a connected database from the database `_User` table
3. `SET-DB-CLIENT` statement, which asserts a user ID that is authenticated to a trusted domain registry (database- or application-defined, depending on your configuration) and asserts that ID as the connection ID for a specified connected database using a sealed client-principal object
4. `SECURITY-POLICY:SET-CLIENT( )` method, which asserts a user ID that is authenticated to a trusted domain registry (database- or application-defined, depending on your configuration) and asserts that ID as the connection ID for all connected databases that do not already have an explicitly set database connection ID using a sealed client-principal object

If you set the database connection ID with one of the first three elements, using the SET-CLIENT( ) method has no effect on this setting. Whatever way you set the database connection ID, the act of doing so generates an appropriate audit event (if auditing is enabled), and the resulting database connection ID, for each database where auditing is enabled, becomes the auditing ID.

The configured auditing identity, itself, has no effect on database authorization. OpenEdge authorizes all run-time CAN* permissions and auditing privileges on a database, as well as database connections, using the database connection ID. So, when you take the default and specify the database connection ID as the auditing ID, it happens that the same user ID used to authorize database access is also associated with the audit event records generated in a given audit-enabled database.

Note: CAN* permissions refer to the permissions for modifying tables and fields that you can set for each user in the OpenEdge Data Administration tool or character-mode Data Dictionary. Auditing privileges refer to permissions to perform auditing operations. For more information on CAN* permissions settings, see *Manage ABL Applications* and Application Security on page 97. For more information on auditing privileges, see Configuring auditing privileges on page 245.

Using the OpenEdge session ID

When you configure auditing to use the OpenEdge session ID as the auditing ID, you enable all audit-enabled databases so-configured to use a single auditing ID authenticated from a single authentication system. You can assert a single OpenEdge session ID as the auditing ID using the SECURITY-POLICY:SET-CLIENT( ) method. This method asserts the OpenEdge session ID to OpenEdge and validates it against the application trusted domain registry using a sealed client-principal object. Setting the OpenEdge session ID does not, in itself, generate an audit event. The process of initiating and managing a client login session for an OpenEdge session ID can, however, set several different auditing events. For more information, see Managing audit event context on page 248.

The configured auditing identity, itself, has no effect on database authorization. OpenEdge authorizes all run-time CAN* permissions and auditing privileges on a database, as well as database connections, using the database connection ID. When you use the OpenEdge session ID as the auditing ID, the user ID (database connection ID) used to authorize database access and audit privileges might not have the same value as the user ID (OpenEdge session ID) that is associated with the audit event records generated in a given audit-enabled database.

Note: CAN* permissions refer to the permissions for modifying tables and fields that you can set for each user in the OpenEdge Data Administration tool or character-mode Data Dictionary. Auditing privileges refer to permissions to perform auditing operations. For more information on CAN* permissions settings, see *Manage ABL Applications*. For more information on auditing privileges, see Configuring auditing privileges on page 245.

Application design considerations

For legacy applications that use the database _User table to authenticate users, you might not require any database configuration or coding changes to set up auditing security. Your existing usage of the User ID (-U)/Password (-P) or the SETUSERID function might be sufficient. This is especially true in a client/server configuration, where the client has direct access to the database and the application and database user are identical. However, note that where you connect to multiple databases, each database can generate auditing events associated with a different connection ID. If you want the auditing ID to be the same for all connected databases, you must be sure to authenticate the same database connection ID for each database.

If you want to use a single OpenEdge session ID as your auditing ID, regardless of individual requirements for database connection and access authorization, set the following database options:

- **Use Application User Id for Auditing** — Described in an earlier section, this tells the database to use the OpenEdge session ID as the auditing ID.

- **Record Authenticated Client Sessions** — This option ensures that client login session context for the OpenEdge session ID is recorded as part of the audit trail. For more information on using client login sessions with auditing, see Managing audit event context on page 248.

To provide an OpenEdge session ID for your application, add the ABL code necessary to assert an OpenEdge session ID using the client-principal object and the `SECURITY-POLICY:SET-CLIENT( )` method. For more information, see Application Security on page 97.

Many legacy applications use their own ABL authentication systems to enforce a single application user ID. If you want to use this application user ID as your auditing ID, you can maintain your existing ABL authentication code. However, you must, again, set the appropriate database options and add the ABL code necessary to assert your application user ID as an OpenEdge session ID using the client-principal object and the `SECURITY-POLICY:SET-CLIENT( )` method, as described in the previous paragraphs.

If you are developing a multi-tier application, especially one that conforms to the OpenEdge Reference Architecture (OERA), you must use the client-principal object and `SECURITY-POLICY` system handle to maintain the correct application user ID and auditing identity between application server clients and the application server. For more information, see Application Security on page 97

Verifying that auditing is enabled

If your application is written to be audited, especially if it generates application events, it is an *audit-aware* application. There is no requirement that any connected database be enabled for auditing in order to run an audit-aware application. However, no audit events, including explicit application events, are generated for the application unless at least one connected database has auditing enabled.

However, if you use auditing, it is typically critical to your application environment. If your application requires auditing to be enabled before it can run, you can verify that this is the case using the `AUDIT-ENABLED` function. This function can verify that at least one of the connected databases in the ABL session is audit-enabled, and you can code a response accordingly.

For more information on enabling a database for auditing, see *Introduction to Security and Auditing*.

Configuring auditing privileges

OpenEdge supports a separate set of auditing privileges to control who can do what to auditing policies and data. To prevent unauthorized access this information, the compile-time and run-time support for `CAN*` permissions on database tables and fields have no effect on the auditing metaschema.

Depending on the function of your auditing-aware application, you must ensure that the users who can run it have the appropriate auditing privileges set. The following table lists the auditing privileges and the application functionality that they support.

Table 31: Auditing privilege capabilities

This auditing privilege allows...	This set of capabilities...
Application Audit Event Inserter (Optional)	Generating application events based on active audit policies
Audit Administrator	Configuring audit policies and reading audit data
Audit Data Archiver	Creating, reading, and deleting audit data
Audit Data Reporter	Reading audit policies and data

OpenEdge enforces the Application Audit Event Inserter privilege only if you set the appropriate database auditing option (**Enforce Audit Insert Privilege**). If you set this option for an audit-enabled database, any user who runs an audit-aware application that generates application events must have the Application Audit Event Inserter privilege, and depending on the application architecture, might also require the Audit Data Reporter privilege. The Audit Administrator and Audit Data Archiver privileges are required by any user who runs an audit configuration and audit archiving tool, respectively. And the Audit Data Reporter privilege is also required for any user who runs an audit reporting tool. For more information on audit privileges, see *Introduction to Security and Auditing*. For more information on setting audit privileges for users, see the Data Administration online help and *Database Tools*.

Recording application events

OpenEdge provides core support for different types of auditing events at three basic levels of auditing:

- **Raw database record** — *Rawdatabase record events* (or *database events*) are OpenEdge-defined events that identify basic database operations (such as reading and writing records) and any attempts to perform such operations by an application. OpenEdge records raw database record events as specified in the active audit policies or as required by the OpenEdge auditing framework.
- **Internal** — *Internal events* are OpenEdge-defined events that identify actions that occur directly or indirectly as a result of session execution (such as authenticating or asserting a user ID). These include actions such as running database utilities, changing the database schema, and so on. OpenEdge records internal events as specified in the active audit policies or as required by the OpenEdge auditing framework.
- **Application** — *Application events* are events that you can define to OpenEdge for the purpose of identifying application-specific actions not otherwise identified by internal or database events (such as reading certain tables or selecting certain menu options). These also include events for adding context to raw database record events. You must invoke the recording of application events at appropriate points in the control flow of your application. OpenEdge then records these events as specified in the active audit policies.

For more information on the various types of audit events, see *Introduction to Security and Auditing*. The following sections describe how you can invoke the recording of application events from within ABL and how to organize all auditing events, including application events, within various auditing contexts.

- Planning application events on page 246
- ABL for generating application events on page 247
- Instructing OpenEdge to record an application event on page 248
- Managing audit event context on page 248
- Application design considerations on page 244

Planning application events

Typically, when you develop your application, you create all of the application events that the application is able to generate. You define what is an auditable event, depending on your application. Generally, an auditable event has two characteristics:

- It represents some operation or point of execution within your application that you want securely and reliably reported.
- There is no OpenEdge-defined raw database record or internal event whose reporting can properly represent the information you want reported.

When you deploy your application, you provide any documentation and tools support necessary for the deployment site to create the audit policies for generating an audit trail with the events you have defined. An audit-enabled database can contain one or more audit policies that specify specific database, internal, and application events (from all available events) that can be included in the audit trail recorded in that database. The actual audit events generated, depend on what audit policies are active. The sum total of active audit policies specify all of the audit events that the application actually generates when the connected database is audit-enabled.

As you design and code the application, you might need to organize auditing events within different auditing contexts that group audit events in different ways. You can specify all of the context information required in your application for organizing audit events, except the transaction context for database events, which is controlled entirely by OpenEdge. For more information on planning application events and their context, see *Introduction to Security and Auditing*.

ABL for generating application events

Many ABL statements and functions can generate database or internal events automatically, based active audit policies. In other words, OpenEdge defines events that certain ABL statements and functions always generate when audit policies specify them to do so. These include statements, for example, that participate in database transactions and that control the assertion of user identities, among others.

ABL also allows you to explicitly generate application events using the `AUDIT-CONTROL` system handle. These include events that you define and OpenEdge-defined events that control audit event context. If database options are set to enforce it, the database connection ID for a database must have the Application Audit Event Inserter privilege in order to generate and record audit events using the `AUDIT-CONTROL` system handle. The following table lists the `AUDIT-CONTROL` handle methods that you can use to explicitly generate events in your application:

Table 32: Methods for generating application events

This AUDIT-CONTROL method...	Provides this function...
`BEGIN-EVENT-GROUP( )`	Starts an event group
`CLEAR-APPL-CONTEXT( )`	Clears the current application context
`END-EVENT-GROUP( )`	Ends the current event group
`LOG-AUDIT-EVENT( )`	Records an application-defined audit event
`SET-APPL-CONTEXT( )`	Sets the current application context

The following sections describe how to code these methods.

Instructing OpenEdge to record an application event

The basic method for generating an application event is the AUDIT-CONTROL:LOG-AUDIT-EVENT( ) method, using the following syntax:

Syntax

```
LOG-AUDIT-EVENT( event-id, event-context  [, event-detail
  [, user-detail  ]  ]  )
```

Invoking this method generates the audit event in any connected and audit-enabled database whose active audit policies also enable this event, identified by its *event-id*. The method returns a character string containing a universally unique identifier (UUID) that serves as a primary index for the generated audit event record. For more information on UUIDs, see the sections on managing identities in Application Security on page 97.

You can specify any integer value from 32000 or greater as the *event-id*. The remaining values can specify any character strings that you find helpful in describing this event. This information is designed to indicate, respectively, some context information for this instance of the event (*event-context*, which is an alternate index for querying the audit event record), any more detailed information about the event (*event-detail*), and any information about the application user associated with the auditing ID for the event (*user-detail*), which you might obtain from attributes of the client-principal object.

Note: As you invoke this method, you might keep notes on the values you set to help provide documentation for creating the event and audit policies for event in the database at application deployment sites.

The following code fragment shows how you might code this method:

```
DEFINE VARIABLE iEmployeeReadEvent AS INTEGER   NO-UNDO INITIAL 32000.
DEFINE VARIABLE caEventPrimaryKey  AS CHARACTER NO-UNDO EXTENT 100.
DEFINE VARIABLE iEventIdx          AS INTEGER   NO-UNDO.
...
iEventIdx = iEventIdx + 1.
caEventPrimaryKey[iEventIdx] =  AUDIT-CONTROL:LOG-AUDIT-EVENT(iEmployeeReadEvent, "First
 record read.").
```

This example assembles a list of generated application events in order of occurrence by storing the generated UUID returned for each event in an array for later reference.

Managing audit event context

OpenEdge provides an architecture for adding context to the audit event records in an audit trail. Without audit event context, an audit trail contains a stream of events that have little apparent relationship to one another. Using various audit contexts, you can group audit events together in various combinations. Then, when you want to generate auditing reports, you can report on audit events based on the context or contexts in which they are recorded.

OpenEdge can organize auditing events within the following different event context areas:

- Database transactions
- Client login sessions

- Application context
- Application event groups

You can explicitly control all of these event contexts, except the database transaction context. OpenEdge always generates database transaction context information (transaction ID and sequence values) that is recorded with each auditing event generated during a database transaction. Otherwise, you can generate context information for the remaining context areas that also appears in audit event records using the ABL elements described in this section.

Otherwise, the OpenEdge auditing context architecture allows you to nest these context areas in different ways to organize the auditing events generated by your application. For more information on this auditing context architecture, see *Introduction to Security and Auditing*. The following sections describe how to generate context information for client login sessions, application context, and application event groups.

Generating client login session context

Client login session context information consists of a client session ID that is recorded for each audit event generated during a client login session. Optionally, it also includes a record keyed by this client session ID that describes the authentication information for the client login session. OpenEdge supports two types of authentication systems for authenticating a user:

- The database `_User` table internal to the OpenEdge RDBMS.
- An ABL application-managed (external) authentication system that can optionally have a corresponding authentication domain defined for it in a trusted domain registry, either configured in an OpenEdge RDBMS or built at run time by the application itself. While not required to log in a client login session, this trusted domain registry is used to set user identities using the `SET-CLIENT( )` method or the `SET-DB-CLIENT` built-in function.

OpenEdge allows you to use both types of authentication system to authenticate a user ID (auditing ID) that is recorded with all audit event records generated for the specified user ID, and which can also be used to indicate the application user ID at the time the event is recorded. For more information on how to specify a user identity as the auditing identity, see Setting up application security for auditing on page 241.

To maintain an auditing context for a client login session, you must do the following:

- In your application ABL, authenticate the user ID that you configure as the auditing ID to an external authentication system, and assert and validate that user ID against the appropriate trusted domain registry using an initialized and sealed client-principal object. For more information, see Setting up application security for auditing on page 241.
- If you want additional information about the client login session recorded for the auditing context, in the Data Administration tool or Data Dictionary (character mode only), select the **Record Authenticated Client Sessions** option in the **Database Options** dialog box. For more information, see the Data Administration online help or *Database Tools*. Setting this option allows a client login session record (`_client-session`) to be written to the database that describes information about the authentication domain for the user ID represented by the client-principal object, various session security values (such as the database unique identifier), and additional detail from selected client-principal object attributes. When auditing is also enabled, this record becomes part of the auditing context.

The initiation and management of the client login session context is primarily the responsibility of methods on the client-principal object, which log in and manage a client login session. Without a client-principal object (using the SETUSERID function or User ID (-U)/Password (-P) parameters), OpenEdge records the configured auditing ID in all related audit event records, but creates no *auditing client login session* (client login session for the auditing ID). However, when you use a client-principal object to manage an auditing client login session, OpenEdge generates audit events to track the session, such as login, logout, and even the failure to log in (if so directed). Logging in with a client-principal object also causes the value of its SESSION-ID attribute to be written to every audit event record related to this auditing client login session. Again, the associated auditing ID (USER-ID attribute on the associated client-principal object) is written to all audit event records while the auditing client login session is active (logged in) and the auditing ID is also set as the client login session user ID.

Note: Each audit enabled database connected to an OpenEdge session can maintain one current database connection ID, which can be set for a client login session established using its client-principal object.

OpenEdge supports several ABL elements that provide user identity and client session context information for the audit trail. The client-principal object provides most of the client login session context information. The following table shows ABL elements, listed in general order of usage, that generate auditing events or otherwise provide information for an auditing client login session context.

Table 33: ABL elements for auditing client login session context

ABL element	Description
SESSION-ID	Client-principal object attribute that specifies the client login session ID used to identify the client login session context in the audit trail. This value also keys any _client-session record created for the login session.
AUDIT-EVENT-CONTEXT	Client-principal object attribute that allows you to provide additional information to the audit event record's _Event-context field, which you can use to later to query audit event records. This value is written to audit event records generated for the AUTHENTICATION-FAILED( ), SEAL( ), and LOGOUT( ) methods.
AUTHENTICATION-FAILED( )	Client-principal object method that generates an audit event for a failed user login.
SEAL( )	Client-principal object method that generates an audit event for a successful user login, and also generates an optional client login session record with additional information.
SET-CLIENT( )	SECURITY-POLICY system handle method that sets the default OpenEdge session ID (which can be the auditing ID) using a client-principal object instance. It also asserts database connect IDs similar to the SET-DB-CLIENT function.

ABL element	Description
SET-DB-CLIENT	ABL function that asserts a user ID as the database connection ID (which can be the auditing ID) using a client-principal object instance. This function also generates an audit event for asserting the connection ID for a particular database.
LOGOUT( )	Client-principal object method that generates an audit event for user logout. It also logs out the client login session and sets the client-principal's LOGIN-STATE attribute to "LOGOUT" so that it cannot be validated to set an OpenEdge session or database connection ID.

If the option to record authenticated client sessions is set for an audit-enabled database, calling the SEAL( ) method on an associated client-principal object both logs in the auditing client session (generating an audit event record) and writes a _client-session record. In addition, OpenEdge relates every audit event record generated while this client login session user ID is set as the current auditing ID (including the login audit event record) to this _client-session record by the value of the SESSION-ID attribute on the client-principal object.

So, all audit event records generated, while the user ID for a given client login session is set as the current auditing ID, share the same session ID value and user ID value (which is also the configured auditing ID), and they also share these values with any _client-session record (related by the session ID value) that happens to be written for the given auditing client login session context.

For more information and examples of using the ABL elements in the above table to maintain client login sessions, see Application Security on page 97.

Generating application context

Application context allows you to define your own criteria for organizing audit events together. So, for example, you might associate all audit events in the same application context that are generated for an inventory module to help distinguish them from events that are generated for a shipping and receiving module; or you might want to track access to specific windows and controls in a user interface; or you might have a ProDataSet involved in transactions to update multiple databases, and you want to associate the change tracking in that ProDataSet with the appropriate database. Again, you can use the resulting context to structure the reporting of audit events generated in the resulting audit trail.

OpenEdge generates a new application context event record with each call to the SET-APPL-CONTEXT( ) method on the AUDIT-CONTROL system handle. When you want to end the scope of the current application context, call the CLEAR-APPL-CONTEXT( ) method. The syntax for these methods follows:

Syntax

```
ctx-id = AUDIT-CONTROL:SET-APPL-CONTEXT
                    (audit-event-ctx [ , event-detail [ , user-data ] ] )
```

```
logical = AUDIT-CONTROL:CLEAR-APPL-CONTEXT( )
```

The *ctx-id* is an application context ID that is a UUID returned as a Base64 character value (with pad characters removed). This application context ID is recorded in all audit event records until the context is changed or cancelled. Each invocation of the SET-APPL-CONTEXT( ) method generates a new application context ID that replaces the previous one in all subsequent audit event records. Thus, only one application context can be active at a time and no nesting of application context is possible.

The input parameters to SET-APPL-CONTEXT( ) are all character values that allow you to specify different levels and types of detailed information about the context, as appropriate for the application. The first, required, parameter specifies an alternate index to the audit event record. So, its length is limited to the maximum size of an index value. Thus, you can use this value as a key for reporting purposes.

If you want to stop recording any application context ID values in audit event records, you must invoke the CLEAR-APPL-CONTEXT( ) method. You can test its logical value for a successful invocation. At no time does the ABL session ever implicitly invoke the CLEAR-APPL-CONTEXT( ) method. Therefore, you must ensure that all code paths (especially error handling code paths) end the application context correctly. You can also begin another application context at any point after clearing the current application context.

If at any point in the ABL session you need to determine the current application context ID, you can examine the value of the APPL-CONTEXT-ID attribute on the AUDIT-CONTROL system handle. This attribute returns the Unknown value (?) if no application context is currently set.

The following code fragments show use of these methods and attributes:

```
DEFINE VARIABLE user-task AS CHARACTER NO-UNDO.
DEFINE VARIABLE id        AS CHARACTER NO-UNDO.
...
user-task = "Salary adjustments".
...
id = AUDIT-CONTROL:SET-APPL-CONTEXT
  ("Payroll app", "FICA calculation", user-task).
...
AUDIT-CONTROL:CLEAR-APPL-CONTEXT( ).
...
IF id <> AUDIT-CONTROL:APPL-CONTEXT-D THEN ...
```

Generating audit event groups

Audit event groups work in a similar fashion to application context. You can also nest a series of application contexts within an audit event group or nest a series of audit event groups within an application context.

OpenEdge generates a new audit event group record with each call to the BEGIN-EVENT-GROUP( ) method on the AUDIT-CONTROL system handle. When you want to end the scope of the current audit event group, call the END-EVENT-GROUP( ) method.

The syntax for these methods follows:

Syntax

```
grp-id = AUDIT-CONTROL:BEGIN-EVENT-GROUP
                    (audit-event-ctx [ , event-detail [ , user-data ] ] )
```

```
logical = AUDIT-CONTROL:END-EVENT-GROUP( )
```

The *grp-id* is an audit event group ID that is a UUID returned as a Base64 character value (with pad characters removed). This audit event group ID is recorded in all audit event records until the event group context is changed or ended. Each invocation of the BEGIN-EVENT-GROUP( ) method generates a new audit event group ID that replaces the previous one in all subsequent audit event records. Thus, only one audit event group can be active at a time and no nesting of audit event groups is possible.

The input parameters to BEGIN-EVENT-GROUP( ) are all character values that allow you to specify different levels and types of detailed information about the audit event group context, as appropriate for the application. The first, required, parameter specifies an alternate index to the audit event record. So, its length is limited to the maximum size of an index value. Thus, you can use this value as a key for reporting purposes.

If you want to stop recording any audit event group ID values in audit event records, you must invoke the END-EVENT-GROUP( ) method. You can test its logical value for a successful invocation. At no time does the ABL session ever implicitly invoke the END-EVENT-GROUP( ) method. Therefore, you must ensure that all code paths (especially error handling code paths) end the audit event group correctly. You can also begin another audit event group at any point after ending the current audit event group.

If at any point in the ABL session you need to determine the current audit event group ID, you can examine the value of the EVENT-GROUP-ID attribute on the AUDIT-CONTROL system handle. This attribute returns the Unknown value (?) if no audit event group context is currently active.

The following code fragments show use of these methods and attributes, in this example, creating an audit event group with nested application context:

```
DEFINE VARIABLE user-task AS CHARACTER NO-UNDO.
DEFINE VARIABLE ctx-id    AS CHARACTER NO-UNDO.
DEFINE VARIABLE grp-id    AS CHARACTER NO-UNDO.
...
user-task = "Salary adjustments".
...
grp-id = AUDIT-CONTROL:BEGIN-EVENT-GROUP
  ("Payroll app", "tax calculations", user-task).
...
ctx-id = AUDIT-CONTROL:SET-APPL-CONTEXT
  ("Payroll app", "Federal tax calculation", user-task).
...
AUDIT-CONTROL:LOG-AUDIT-EVENT (34122, ctx-data).
...
ctx-id = AUDIT-CONTROL:SET-APPL-CONTEXT
  ("Payroll app", "FICA calculation", user-task).
...
AUDIT-CONTROL:LOG-AUDIT-EVENT (34123, ctx-data).
...
AUDIT-CONTROL:CLEAR-APPL-CONTEXT.
...
AUDIT-CONTROL:END-EVENT-GROUP.
...
IF grp-id <> AUDIT-CONTROL:EVENT-GROUP-D THEN ...
```

In this example, two application contexts are invoked in sequence for a different series of audit events, and both are nested within the audit event group. You can also invert this nesting, with a single application context nesting a series of audit event groups. The choice depends entirely upon your application design preferences.

Application design considerations

For any client/server application, where the client is accessing databases directly, generating application events is straight forward. You invoke the appropriate methods on the AUDIT-CONTROL system handle and OpenEdge ensures that the corresponding audit event records are recorded to every connected and audit-enabled OpenEdge RDBMS.

However for a multi-tier application, especially an OERA-conformant application, where the application server client does not access any data sources directly, you cannot execute application event methods directly on the client. Especially for a WebClient, because there is no connected database, these methods execute without effect.

Supporting OERA-conformant clients

To support auditing for an OERA-conformant client, you must create an API on the application server especially for generating application events on behalf of the client. The granularity of this API from the client perspective is entirely application-dependent. You can provide remote procedure or user-defined function calls, perhaps parameterized, that generate one audit event at a time to reflect client-specific conditions, or you can simply hide `AUDIT-CONTROL` method calls inside task-oriented application service calls, such as inventory look ups or other application-specific operations.

If you require a fine-grained application event API to audit client-specific conditions, you might also need to examine active audit policies to identify active application events that the API can generate for specific client calls. It all depends on how general your application event API must be to cover the application functionality.

Examining active audit policies

To examine active audit policies in the application server session, you must ensure that the database connection ID for a given audit-enabled database has the Audit Data Reporter or Audit Administrator privilege set for it. You can then access the active audit policies in the following ways:

- Directly access audit policy tables in each connected and audit-enabled OpenEdge RDBMS
- Access appropriate query procedures supported by the OpenEdge Audit Policy Maintenance APIs

The Audit Policy Maintenance APIs represent a set of ABL persistent and non-persistent procedures that allow you to query audit policies or perform audit policy maintenance tasks, depending on your audit privileges.

For purposes of querying active audit policies, you can use the following procedures from these APIs:

- `get-policies-merge` internal procedure of the generic utility API (`_aud-utils.p` persistent procedure). This procedure returns data about active policies only. However, you must use other procedures in the API to load a ProDataSet with audit policy data from audit-enabled databases before you can access the active policy information.
- `_get-policies.p` non-persistent procedure. This procedure loads all of the audit policies in a given audit-enabled database into a ProDataSet. You can then query the ProDataSet for active audit policy data. If you want to query more than one database, you must invoke the procedure for each one.

These procedures are all located in the following OpenEdge installation location:

```
OpenEdge_install_dir\src\auditing\
```

For more information on using the Audit Policy Maintenance APIs and the audit policy tables, see Custom audit configuration tools on page 255. For more information on the Audit Policy Maintenance APIs themselves, see Audit Policy Maintenance APIs on page 617.

Once you have identified auditable application events, as specified in active audit policies, you can invoke the `AUDIT-CONTROL:LOG-AUDIT-EVENT( )` method for an appropriate application event in response to a client call to your own application event API.

Developing auditing tools

Auditing tools support the following auditing management functions:

- **Audit configuration** — Creating and maintaining the audit policies that determine what is or can be audited in an application
- **Audit archiving** — Moving audit data (audit trails) from short-term, live application storage, to long-term archival storage
- **Audit reporting** — Generating human-readable reports on existing audit data and policies

OpenEdge provides general purpose auditing tools to perform the following auditing management functions:

- Auditing configuration using the OpenEdge Audit Policy Maintenance tool
- Audit archiving using the OpenEdge Audit Archiving utility
- Audit reporting using the Auditing Reports available from the Data Administration tool and character-mode Data Dictionary

---

Note: The Auditing Reports available with OpenEdge report on internal OpenEdge-defined events only.

For more information on these OpenEdge auditing tools, see *Introduction to Security and Auditing*.

However, you might want to write your own tools to accomplish these tasks. Each of the OpenEdge solutions has requirements or limitations that might not be appropriate for your auditing environment. For audit configuration and archiving, you might find it necessary to build your own tools in order to support specific input/output formats, to handle special audit policy configuration relationships, or to archive audit data in a different way.

The following sections describe OpenEdge support and basic requirements to develop the following custom auditing tools in ABL.

- Custom audit configuration tools on page 255
- Custom audit archiving tools on page 257

For reporting, it is impossible to anticipate all possible query requirements that you might have in a given environment. For an overview of audit query issues and where to find more information about writing custom audit reporting tools, see *Introduction to Security and Auditing*.

Custom audit configuration tools

OpenEdge provides core support for managing audit policies (rules) that govern how audit trails are defined, tracked, and recorded. The Audit Policy Maintenance tool provided with OpenEdge allows you to:

- Create, delete, and configure audit policies
- Create audit policy templates to simplify initial audit configuration
- Import and export audit policies in order to apply them to multiple databases
- Define application events generated in your applications

For more information on configuring audit policies using Audit Policy Management, see *Introduction to Security and Auditing* and the online help for the Audit Policy Maintenance tool.

Accessing the audit policy tables

You can write your own tools to perform same functions supported by the Audit Policy Maintenance tool. To do so, you must:

- Write the ABL code to query and update audit policy tables in a given audit-enabled database.
- Use methods of the `AUDIT-POLICY` system handle to manage the effect of any changes to existing auditing policy settings.

The following tables store the basic audit policy settings for an audit-enabled database:

- **`_aud-event`** — Defines the supported audit events. Each OpenEdge-defined event has a unique event ID (`_Event-id` field value) with a value less than 32000. All application-defined events must also have a unique event ID with a value of 32000 or greater.
- **`_aud-audit-policy`** — Provides a mechanism to define named audit policies configured according to different policy requirements.
- **`_aud-event-policy`** — Defines policy settings for audit events of a specific named audit policy, and specifies such information as if the audit event is enabled and the level of audit detail to be recorded.
- **`_aud-field-policy`** — Optionally, defines any field-level policy settings for a specific named audit policy. Fields without settings inherit any settings for the table, if any, and are disabled from auditing otherwise.
- **`_aud-file-policy`** — Optionally, defines any table-level policy settings for a specific named audit policy. Tables without settings are disabled from auditing.

In addition to these principle audit policy tables, the following tables provide information about auditing and security features of a given OpenEdge RDBMS. You do not typically change these tables for Audit Policy Maintenance activities, but they might be useful for reference purposes:

- **`_db`** — Provides standard metaschema information in an OpenEdge RDBMS, including a global unique identifier (GUID) that is used to uniquely identify the database in audit data aggregated from multiple databases.
- **`_db-option`** — Provides an extensible means to define criteria for handling various database management options, including auditing and security. OpenEdge provides such options for managing auditing and general database security identities and permissions.
- **`_db-detail`** — Stores auditing-specific information about a database, including the message authentication code (MAC) key used to secure and seal audit data, depending on the audit data security level (an audit policy setting).

For more information on these tables, see the sections on audit policies and data in *Introduction to Security and Auditing*.

In order to allow audit policy changes to occur without taking an audit-enabled database off line, ABL provides a mechanism to inform the database server that audit policy changes have occurred so it can refresh its audit policy settings. You can do this using the `REFRESH-AUDIT-POLICY( )` method on the `AUDIT-POLICY` system handle. By invoking this method for a specified database, the OpenEdge RDBMS can immediately use the latest changes to audit policies in order to process and record audit events.

Accessing the Audit Policy Maintenance APIs

The rules for configuring and managing audit policies directly in the audit policy tables are complex. As an aid to following these rules, OpenEdge provides a set of Audit Policy Maintenance APIs that you can use to properly manage these tables. These APIs rely on standard data definitions for ABL temp-tables and ProDataSets that are used as intermediate audit policy storage. They also support remote client access to API services hosted on an application server.

OpenEdge uses these APIs to implement the Audit Policy Maintenance tool, and the source code for the tool is installed with OpenEdge. You can thus use this tool as a working sample application for writing your own audit policy configuration tools. You can find all the source code for the APIs, including the Audit Policy Maintenance main procedure (`_apmt.p`) in the following OpenEdge installation directory:

```
OpenEdge_install_dir\src\auditing
```

For a description of the individual API procedure files, see Audit Policy Maintenance APIs on page 617.

Audit configuration security

When accessing the audit policy tables directly or through the Audit Policy Maintenance APIs, the user must have the Audit Administrator privilege in order to create, read, update, and delete audit policy records. To query the audit policy tables, the user only requires the Audit Data Reporter privilege.

Custom audit archiving tools

OpenEdge provides core support for archiving audit data from short-term audit-trail storage to long-term audit-trail storage. The Audit Archiving utility provided with OpenEdge allows you to:

- Archive audit data (move selected audit data records and copy other supporting records) from active short-term storage in an OpenEdge RDBMS to long-term storage in the form of audit binary dump (`.adb`) files.
- Calculate and store an HMAC data integrity value along with the archived audit data that can be used to verify that the data has not been tampered with before it is loaded into another OpenEdge RDBMS. (An HMAC is a form of message authentication code that uses a secret key.)
- Purge audit data (delete selected audit data records) from the short term storage with or without an archive of the same data.
- Load audit data contained in `.adb` files into an audit-enabled OpenEdge RDBMS.
- Run multiple concurrent instances of the utility, which handle record access conflicts through a record locking protocol. This same record locking protocol allows you to run the utility on a live database without the need to take it offline.

For more information on the OpenEdge Audit Archiving utility, see *Introduction to Security and Auditing*.

Accessing the audit data and supporting tables

You can write your own tools to archive audit data in other ways than are supported by the OpenEdge Audit Archiving utility. For example, where the OpenEdge Audit Archiving utility dumps audit data to `.adb` files, you might want to move the data directly from the database used for short-term storage directly to a database that you use for long-term storage. For another example, the Audit Archiving utility never deletes the audit archive events generated during the process of archiving the audit data in a given database. You might want your custom audit archiving tool to delete the audit archiving events that are generated in short-term storage during the process of archiving.

Caution: Archiving audit data requires special privileges because it is the only auditing activity that allows audit data to be programmatically created and deleted. Maintain appropriate security for any tool that you build to do audit archiving. For more information, see Audit archiving security on page 259

The tables for auditing archiving include:

- **_aud-audit-data** — Contains the audit records generated as a result of enabling audit events in an active audit policy. For archiving, you generally move the records in this table from short-term to long-term storage.
- **_aud-audit-data-value** — Optionally, contains the before and after field values for database events that modify fields in one record per modified field. For archiving, you generally move the records in this table from short-term to long-term storage.
- **_aud-event** — Defines the audit events that generate the audit records in _aud-audit-data. For archiving, you generally copy (**not** move) the records to long-term storage for audit events represented by the generated audit records.
- **_client-session** — Optionally, records information about the client login session, including how the login session was authenticated, in order to support nonrepudiation of the audit data. For archiving, you generally copy (**not** move) the _client-session records to long-term storage that are associated with the generated audit records you are archiving.

  Note: Records for a client session are only available if you specify the option to record client sessions in the database whose short-term audit data you are archiving. For more information on recording client sessions, see Generating client login session context on page 249.

- **_db-detail** — Contains audit data about the database being archived, that is, the database where the audit trail being archived was originally generated. For archiving, you generally copy the _db-detail records to long-term storage that are associated with the generated audit records you are archiving. There can be more than one _db-detail record if you are archiving audit data from more than one audit-enabled database. This provides access to the MAC key (database passkey) used to validate audit data record seals from each database that provides audit data records for archiving. For more information, see Audit archiving security on page 259.

For more information on these tables, see the sections on the audit data tables in *Introduction to Security and Auditing*.

Handling archiving requirements

The exact archiving requirements for your custom audit archiving utility are largely application dependent. However, there are a few issues that you must consider when writing the code for such a utility:

- You must manage any record access conflicts that might arise from archiving a live online database or from running multiple audit archiving utilities concurrently on the same set of databases.
- You might have additional security requirements over and above what OpenEdge requires that you want to enforce. For more information, see Audit archiving security on page 259.
- You must manage your own encryption and data integrity if you use your own intermediate storage medium, such as XML. For more information, see the sections on cryptography in Application Security on page 97.
- Some fields in audit data records are stored using Base64 representations of binary bytes strings. You can convert between Base64 character strings and binary byte strings using the BASE64-ENCODE and BASE64-DECODE ABL functions. You can also convert between binary byte strings and strings of hexadecimal character pairs using the HEX-ENCODE and HEX-DECODE ABL functions. For more information, see the sections on managing cryptographic data in Application Security on page 97

Besides any special requirements like these, the archiving procedure generally relies on simple queries of the audit data tables according to your predetermined or user-selectable data selection criteria. Typically, you then write the queried records to your long-term storage in a manner that exactly replicates their content and relationships in the original short-term storage. However, again, your own long-term audit data storage requirements might be different.

Audit archiving security

OpenEdge requires that any user who runs an audit archiving tool, particularly one that deletes and creates audit data records, must have the Audit Data Archiver privilege. You might also authorize users of your own audit archiving utility to have privileged access to the files and directories that constitute your long-term audit data storage or require them to have privileged knowledge of the MAC keys that you use to transport audit data from short-term to long-term storage.

The audit data can be sealed using a MAC key (database passkey), depending on the audit data security level. You can specify this database passkey using the **Admin** > **Database Identification** menu option in the Data Administration tool or character-mode Data Dictionary. An encrypted form of this value is stored in the `_db-detail` table for the database. When you dump and later load the archived audit data, you need this value to ensure the integrity of the data.

You can prompt for the database passkey as part of the archiving process and encrypt it in the same form that is used by OpenEdge and stored in the `_db-detail` table. To encrypt the user-supplied value, pass it to the `ENCRYPT-AUDIT-MAC-KEY( )` method on the `AUDIT-POLICY` system handle. You can then compare this value to the value stored in the `_db-detail` table.

II

Input/Output Processes

For details, see the following topics:

- Handling User Input
- Alternate I/O Sources
- Colors and Fonts
- Creating Reports

6

Handling User Input

These topics explain how you can monitor user input from the keyboard or mouse. Alternate I/O Sources on page 297 explains how to write procedures that accept input from a file.

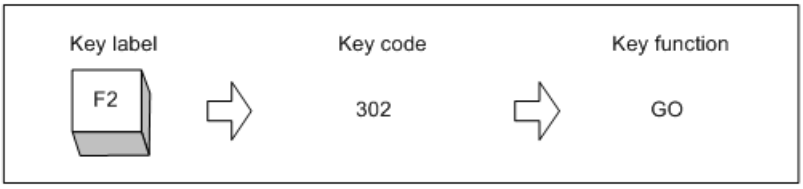
For details, see the following topics:

- The keyboard and the mouse
- Key codes and key labels
- Key functions
- Changing the function of a key
- Using mouse buttons and events
- Telling the AVM how to continue processing
- Monitoring keystrokes during data entry

The keyboard and the mouse

The ABL Virtual Machine (AVM) accepts user input from the keyboard and, where available, from the mouse. Although the keys on different keyboards vary somewhat, ABL (Advanced Business Language) defines several hundred standard key codes that map to common key labels (or sequences). Some of these codes are also mapped to special functions by ABL or the windowing system. As shown in the following figure, the **F2** key label maps to key code 302. On most systems, this key code also maps to the ABL **GO** key function.

Figure 18: Key labels, key codes, and key functions



ABL also defines a four-button mouse model. Each of the four portable buttons maps to a physical button (possibly with a modifying key) on your mouse. For example, the portable SELECT button usually maps to the left button on a two- or three-button mouse. The mouse buttons generate input events that map to key labels. However, the way the AVM handles mouse input is different from the keyboard because of the way the mouse generates events—sending both press (down) and release (up) signals. ABL provides access either to the down or up event or to combinations of these inputs as a single event. For more information on mouse buttons, events, and how to monitor them, see Using mouse buttons and events on page 284.

Key mapping

Your environment determines what key sequences on your keyboard map to each ABL key function. Your environment might reside in the registry (Windows only) or in an initialization file. Examples of initialization files are the `progress.ini` file (in Windows) and the PROTERMCAP file (on UNIX). For more information on environments, see the topics on colors and fonts in this guide and the topics on user interface environments in *Manage ABL Applications*.

Key monitoring

Often, an application does not need to be aware of specific keystrokes or mouse actions by the user because the normal functionality provided by ABL and the window system is sufficient. This is frequently true when you use the procedure-driven programming model. However, you might want to define a special behavior to occur when the user presses a specific key or mouse button. For example, every time the user presses a certain key, you might want to display a message or take some action. This is called *monitoring* the user's keystrokes. ABL provides a set of language constructs to do this, including user interface triggers. For more information, see Monitoring keystrokes during data entry on page 289.

During a session, the AVM responds to key code events according to a precedence that depends on the current user interface and the application design. The following table shows the general order of precedence for ABL key code events.

Table 34: ABL key input precedence

Precedence (high to low)	Key code event
8	In the accelerator list for the current window
7	In the event list for an active user interface trigger

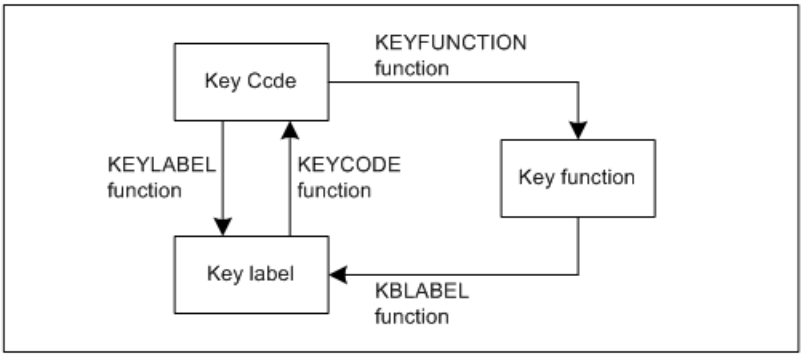
Precedence (high to low)	Key code event
6	An active popup menu key
5	An active menu mnemonic key
4	An active field-level (button) mnemonic key (Windows only)
3	A special ABL internal key (built-in key function, like GO)
2	A key associated with a key function in the PROTERMCAP file (UNIX only)
1	A standard input key ("A", "B", and so on)

Thus, when the user presses a key, the AVM first checks to see if it corresponds to an active accelerator in the current window. If not, the AVM checks for a user interface trigger for that key, then whether the key displays a popup menu, and so on. If the key corresponds to no other function, the AVM processes it as data input, with any required character validation.

Key translation functions

ABL key codes, key labels, and key functions are interrelated, and ABL provides a set of built-in functions to convert from one to another. The following figure shows how the conversions work, and which functions to use for each conversion.

Figure 19: ABL key translation functions



You can use the KEYCODE function to determine the key code for a particular key. For example, the following function returns the value 127, which is the integer code assigned to **DEL**. Use the KEYLABEL function to determine the keyboard label for a particular key code. For example:

```
KEYCODE("del")
```

The following function returns the value **CTRL+G**:

```
KEYLABEL(7).
```

See *ABL Reference* for more information on the KBLABEL, KEYCODE, KEYFUNCTION, and KEYLABEL functions.

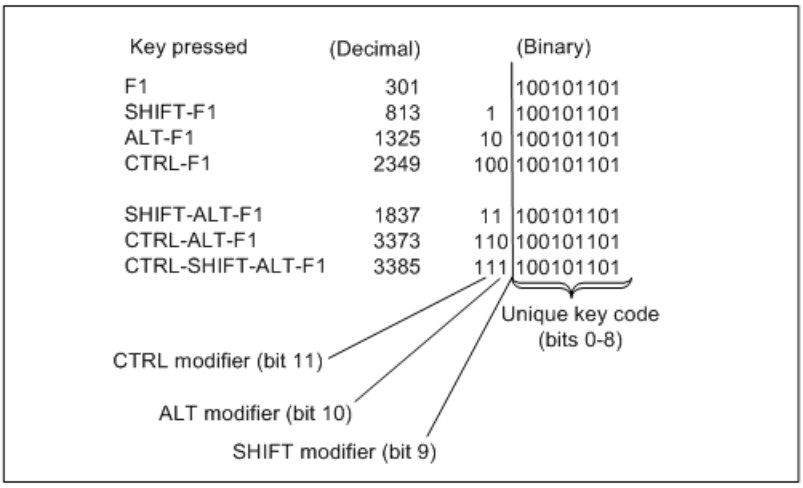
Key codes and key labels

All ABL keys have both a code and a label. The key code is an ABL internal identifier for the key. ABL stores all key codes in 16-bit words. The structure of a key code is as follows:

- **Bits 0–8** — Unique code assigned to the key
- **Bit 9** — The **SHIFT** key modifier (512)
- **Bit 10** — The **ALT** or **ESC** key modifier (1024)
- **Bit 11** — The **CTRL** key modifier (2048)
- **Bits 12–15** — Reserved bits

This structure allows the AVM to respond to a wide variety of key combinations. For example, you can write a trigger that fires when the user presses **CTRL**+**SHIFT**+**ALT**+**F1**. The following figure shows how ABL generates key codes when different keys are pressed.

Figure 20: Key codes



ABL uses this scheme wherever possible. However, some keys are assigned key codes that do not fit into this scheme. The following rules determine what key codes are outside of this scheme:

- If you press **CTRL** plus an alphabetic character or the (@, \,], ^, or _ special characters), the AVM turns the CTRL key modifier off and maps to the appropriate character code. For example, **CTRL+A** is mapped to keycode 1, which accords with the ASCII standard (and therefore most 7–bit character code pages).
- If you press **SHIFT** plus a code page character, the AVM turns off the SHIFT key modifier and maps to the appropriate character code. For example, **SHIFT+A** is equivalent to A, and A is mapped to keycode 65, which accords with the ASCII standard (and therefore most 7–bit character code pages). Also, many code page characters (for example, the comma) do not have uppercase equivalents; therefore, the SHIFT key modifier is turned off.
- If you press **ALT** plus an alphabetic character, the AVM maps to the uppercase alphabetic characters. For example, **ALT+A** and **ALT+a** are both mapped to keycode 1089. The keycode 1089 corresponds to the keycode 65 (A) with the ALT key modifier turned on (65 + 1024 = 1089).

ABL provides more than one key label for certain key codes. The key labels in the following table are the preferred labels, but you can also use the labels in the Table 36: Alternate key codes and key labels on page 276 table.

Table 35: Preferred key codes and key labels

Key code	Key label
0	**CTRL**+@
1 through 7	**CTRL**+**A** through **CTRL**+**G**
8	**BACKSPACE**
9	**TAB**
10 through 12	**CTRL**+**J** through **CTRL**+**L**
13	**ENTER** (Windows) **RETURN** (UNIX)
14 through 26	**CTRL**+**N** through **CTRL**+**Z**
27	**ESC**
28	**CTRL**+****
29	**CTRL**+**]**
30	**CTRL**+**^**
31	**CTRL**+_
32 through 126	Corresponding character
127	**DEL**
128 through 255	Corresponding extended character
300 through 399	**F0** through **F99**
400 through 499	**PF0** through **PF99**
501	**CURSOR**–**UP**
502	**CURSOR**–**DOWN**
503	**CURSOR**–**RIGHT**
504	**CURSOR**–**LEFT**
505	**HOME**
506	**END**
507	**PAGE**–**UP**

Key code	Key label
508	**PAGE–DOWN**
509	**BACK–TAB**
510	**INS**
511	**HELP–KEY**
512	**DELETE**
513	**EXECUTE**
514	**PAGE**
515	**FIND**
516	**INS–LINE**
517	**DEL–LINE**
518	**LINE–ERASE**
519	**PAGE–ERASE**
520	**SHIFT+BACKSPACE**
521	**SHIFT+TAB**
522	**EXIT**
535	**ERASE**
536	**WHITE**
537	**BLUE**
538	**RED**
539	**RESET**
541	**CTRL+BREAK**
551 through 560	**U1** through **U10**
609	**MOUSE–SELECT–UP**
610	**MOUSE–MOVE–UP**
611	**MOUSE–MENU–UP**
612	**MOUSE–EXTEND–UP**

Key code	Key label
617	**MOUSE–SELECT–DOWN**
618	**MOUSE–MOVE–DOWN**
619	**MOUSE–MENU–DOWN**
620	**MOUSE–EXTEND–DOWN**
625	**MOUSE–SELECT–CLICK**
626	**MOUSE–MOVE–CLICK**
627	**MOUSE–MENU–CLICK**
628	**MOUSE–EXTEND–CLICK**
640	**MOUSE–MOVE**
649	**MOUSE–SELECT–DBLCLICK**
650	**MOUSE–MOVE–DBLCLICK**
651	**MOUSE–MENU–DBLCLICK**
652	**MOUSE–EXTEND–DBLCLICK**
812 through 911	**SHIFT+F0** through **SHIFT+F99**
912 through 1011	**SHIFT+ PF0** through **SHIFT+PF99**
1013	**SHIFT+CURSOR–UP**
1014	**SHIFT+CURSOR–DOWN**
1015	**SHIFT+CURSOR–RIGHT**
1016	**SHIFT+CURSOR–LEFT**
1017	**SHIFT+HOME**
1018	**SHIFT+END**
1019	**SHIFT+PAGE–UP**
1020	**SHIFT+PAGE–DOWN**
1021	**SHIFT+BACK–TAB**
1022	**SHIFT+INS**
1023	**SHIFT+HELP–KEY**

Key code	Key label
1024	**ALT+CTRL+@**
1025 through 1031	**ALT+CTRL+A** through **ALT+CTRL+G**
1032	**ALT+BACKSPACE**
1033	**ALT+TAB**
1034 through 1036	**ALT+CTRL+J** through **ALT+CTRL+L**
1037	**ALT+RETURN**
1038 through 1050	**ALT+CTRL+N** through **ALT+CTRL+Z**
1051	**ALT+ESC**
1052	**ALT+CTRL+**
1053	**ALT+CTRL+]**
1054	**ALT+CTRL+^**
1055	**ALT+CTRL+_**
1057 through 1150	**ALT+** corresponding character
1151	**ALT+DEL**
1152 through 1279	**ALT+** corresponding extended character
1324 through 1423	**ALT+F0** through **ALT+F99**
1424 through 1523	**ALT+PF0** through **ALT+PF99**
1525	**ALT+CURSOR–UP**
1526	**ALT+CURSOR–DOWN**
1527	**ALT+CURSOR–RIGHT**
1528	**ALT+CURSOR–LEFT**
1529	**ALT+HOME**
1530	**ALT+END**
1531	**ALT+PAGE–UP**
1532	**ALT+PAGE–DOWN**
1533	**ALT+BACK–TAB**

Key code	Key label
1534	**ALT+INS**
1535	**ALT+HELP–KEY**
1536	**ALT+DELETE**
1537	**ALT+EXECUTE**
1538	**ALT+PAGE**
1539	**ALT+FIND**
1540	**ALT+INS–LINE**
1541	**ALT+DEL–LINE**
1542	**ALT+LINE–ERASE**
1543	**ALT+PAGE–ERASE**
1544	**ALT+SHIFT+BACKSPACE**
1545	**ALT+SHIFT+TAB**
1546	**ALT+EXIT**
1549 through 1558	**ALT+U1** through **ALT+U10**
1559	**ALT+ERASE**
1560	**ALT+WHITE**
1561	**ALT+BLUE**
1562	**ALT+RED**
1563	**ALT+RESET**
1565	**ALT+ctrl+BREAK**
1575 through 1584	**ALT+U1** through **ALT+U10**
1663	**SHIFT+ALT+DEL**
1664 through 1791	**SHIFT+ALT+** corresponding extended character
1836 through 1935	**SHIFT+ALT+F0** through **SHIFT+ALT+F99**
1936 through 2035	**SHIFT+ALT+PF0** through **SHIFT+ALT+PF99**
2037	**SHIFT+ALT+CURSOR–UP**

Key code	Key label
2038	**SHIFT+ALT+CURSOR–DOWN**
2039	**SHIFT+ALT+CURSOR–RIGHT**
2040	**SHIFT+ALT+CURSOR–LEFT**
2041	**SHIFT+ALT+HOME**
2042	**SHIFT+ALT+END**
2043	**SHIFT+ALT+PAGE–UP**
2044	**SHIFT+ALT+PAGE–DOWN**
2045	**SHIFT+ALT+BACK–TAB**
2046	**SHIFT+ALT+INS**
2047	**SHIFT+ALT+HELP–KEY**
2056	**CTRL+BACKSPACE**
2057	**CTRL+TAB**
2075	**CTRL+ESC**
2080 through 2111	**CTRL+** corresponding character
2171 through 2174	**CTRL+** corresponding character
2175	**CTRL+DEL**
2209 through 2303	**CTRL+** corresponding extended character
2348 through 2447	**CTRL+F0** through **CTRL+F99**
2448 through 2547	**CTRL+PF0** through **CTRL+PF99**
2549	**CTRL+CURSOR–UP**
2550	**CTRL+CURSOR–DOWN**
2551	**CTRL+CURSOR–RIGHT**
2552	**CTRL+CURSOR–LEFT**
2553	**CTRL+HOME**
2554	**CTRL+END**
2555	**CTRL+PAGE–UP**

Key code	Key label
2556	**CTRL+PAGE–DOWN**
2557	**CTRL+BACK–TAB**
2558	**CTRL+INS**
2559	**CTRL+HELP–KEY**
2560	**CTRL+DELETE**
2561	**CTRL+EXECUTE**
2562	**CTRL+PAGE**
2563	**CTRL+FIND**
2564	**CTRL+INS–LINE**
2565	**CTRL+DEL–LINE**
2566	**CTRL+LINE–ERASE**
2567	**CTRL+PAGE–ERASE**
2568	**CTRL+SHIFT+BACKSPACE**
2569	**CTRL+SHIFT+TAB**
2570	**CTRL+EXIT**
2573	**CTRL+SHIFT+RETURN**
2583	**CTRL+ERASE**
2584	**CTRL+WHITE**
2585	**CTRL+BLUE**
2586	**CTRL+RED**
2587	**CTRL+RESET**
2559 through 2608	**CTRL+U1** through **CTRL+U10**
2860 through 2859	**CTRL+SHIFT+F0** through **CTRL+SHIFT+F99**
2960 through 3059	**CTRL+SHIFT+PF0** through **CTRL+SHIFT+PF99**
3061	**CTRL+SHIFT+CURSOR–UP**
3062	**CTRL+SHIFT+CURSOR–DOWN**

Key code	Key label
3063	**CTRL+SHIFT+CURSOR–RIGHT**
3064	**CTRL+SHIFT+CURSOR–LEFT**
3065	**CTRL+SHIFT+HOME**
3066	**CTRL+SHIFT+END**
3067	**CTRL+SHIFT+PAGE–UP**
3068	**CTRL+SHIFT+PAGE–DOWN**
3069	**CTRL+SHIFT+BACK–TAB**
3070	**CTRL+SHIFT+INS**
3071	**CTRL+SHIFT+HELP–KEY**
3104 through 3135	**CTRL+ALT+** corresponding character
3163	**CTRL+ALT+[**
3164	**CTRL+ALT+**
3165	**CTRL+ALT+]**
3166	**CTRL+ALT+^**
3167	**CTRL+ALT+_**
3168	**CTRL+ALT+‘**
3195 through 3198	**CTRL+ALT+** corresponding character
3199	**CTRL+ALT+DEL**
3233 through 3327	**CTRL+ALT+** corresponding extended character
3372 through 3471	**CTRL+ALT+F0** through **CTRL+ALT+F10**
3472 through 3571	**CTRL+ALT+PF0** through **CTRL+ALT+PF99**
3573	**CTRL+ALT+CURSOR–UP**
3574	**CTRL+ALT+CURSOR–DOWN**
3575	**CTRL+ALT+CURSOR–RIGHT**
3576	**CTRL+ALT+CURSOR–LEFT**
3577	**CTRL+ALT+HOME**

Key code	Key label
3578	**CTRL+ALT+END**
3579	**CTRL+ALT+PAGE–UP**
3580	**CTRL+ALT+PAGE–DOWN**
3581	**CTRL+ALT+BACK–TAB**
3582	**CTRL+ALT+INS**
3583	**CTRL+ALT+HELP–KEY**
3585	**CTRL+ALT+EXECUTE**
3586	**CTRL+ALT+PAGE**
3587	**CTRL+ALT+FIND**
3588	**CTRL+ALT+INS–LINE**
3589	**CTRL+ALT+DEL–LINE**
3590	**CTRL+ALT+LINE–ERASE**
3591	**CTRL+ALT+PAGE–ERASE**
3592	**CTRL+SHIFT+ALT+BACKSPACE**
3593	**CTRL+SHIFT+ALT+TAB**
3594	**CTRL+ALT+EXIT**
3597	**CTRL+SHIFT+ALT+RETURN**
3607	**CTRL+ALT+ERASE**
3608	**CTRL+ALT+WHITE**
3609	**CTRL+ALT+BLUE**
3610	**CTRL+ALT+RED**
3611	**CTRL+ALT+RESET**
3623 through 3632	**CTRL+ALT+U1** through **CTRL+ALT+U10**
3884 through 3983	**CTRL+SHIFT+ALT+F0** through **CTRL+SHIFT+ALT+F99**
3994 through 4083	**CTRL+SHIFT+ALT+PF0** through **CTRL+SHIFT+ALT+PF99**
4085	**CTRL+SHIFT+ALT+CURSOR–UP**

Key code	Key label
4086	**CTRL+SHIFT+ALT+CURSOR–DOWN**
4087	**CTRL+SHIFT+ALT+CURSOR–RIGHT**
4088	**CTRL+SHIFT+ALT+CURSOR–LEFT**
4089	**CTRL+SHIFT+ALT+HOME**
4090	**CTRL+SHIFT+ALT+END**
4091	**CTRL+SHIFT+ALT+PAGE–UP**
4092	**CTRL+SHIFT+ALT+PAGE–DOWN**
4093	**CTRL+SHIFT+ALT+BACK–TAB**
4094	**CTRL+SHIFT+ALT+INS**
4095	**CTRL+SHIFT+ALT+HELP–KEY**

The above table shows the key labels returned for each key code by the KEYLABEL function. In addition to these key labels, some key codes have alternative labels. Although the KEYLABEL function does not return these values, if you can pass any of these labels to the KEYCODE function, the corresponding key code is returned. The following table lists the alternate key labels.

Table 36: Alternate key codes and key labels

Key code	Alternate key label
7	**BELL**
8	**BS**
10	**LINEFEED, LF**
12	**FORMFEED, FF**
13	**RETURN** (Windows) **ENTER** (UNIX)
27	**ESCAPE**
127	**CANCEL**
501	**UP, UP–ARROW**
502	**DOWN, DOWN–ARROW**
503	**RIGHT, RIGHT–ARROW**

Key code	Alternate key label
504	**LEFT, LEFT–ARROW**
505	**ESC H**
507	**PGUP, PREV–PAGE, PREV–SCRN**
508	**PGDN, NEXT–PAGE, NEXT–SCRN**
509	**SHIFT+TAB** (UNIX) **BACK–TAB** (Windows)
510	**INSERT, INS–CHAR, INS–C, INSERT–HERE**
512	**DEL–CHAR, DELETE–CHAR, DEL–C**
516	**INS–L, LINE–INS**
517	**DEL–L, LINE–DEL**

Key functions

The following table shows the keys that ABL maps to each key function in each user interface. Note that graphical interfaces that have full mouse control do not require as many navigation keys. Note also that in character interfaces, the precise mapping depends on the terminal type, and that UNIX in particular provides many terminal types. The following table lists possible mappings.

Table 37: ABL key functions

Key function	Key label		
	Windows graphical interface	**UNIX character interface**	**Windows character interface**
ABORT	–	**CTRL+**	–
APPEND–LINE	–	**CTRL+A**	**CTRL+A**
BACK–TAB	**SHIFT+TAB**	**CTRL+U**	**CTRL+U** **SHIFT+TAB**
BACKSPACE	**BACKSPACE**	**BACKSPACE** **CTRL+H** **DEL–CHAR**	**BACKSPACE** **CTRL+H**
BELL	**CTRL+G** **BELL**	**BELL**	–

Key function	Key label		
	Windows graphical interface	UNIX character interface	Windows character interface
BLOCK	–	CTRL+V	CTRL+V
BOTTOM–COLUMN	–	ESC CTRL+B	–
BREAK–LINE	–	ESC B	–
CANCEL–PICK	–	ESC CTRL+X	–
CHOICES	–	ESC BACKSPACE ESC CTRL+H	–
CLEAR	–	F8 CTRL+Z	F8 CTRL+Z
CLOSE	F8	ESC Z F8	CTRL+ALT+Z F8
COMPILE	SHIFT+F2	ESC P	CTRL+ALT+P
COPY	–	F11 ESC C	F11 CTRL+ALT+C
CURSOR–DOWN	CURSOR–DOWN	CURSOR–DOWN CTRL+J	CURSOR–DOWN CTRL+J
CURSOR–LEFT	CURSOR–LEFT	CURSOR–LEFT CTRL+O	CURSOR–LEFT CTRL+O
CURSOR–RIGHT	CURSOR–RIGHT	CURSOR–RIGHT CTRL+L	CURSOR–RIGHT CTRL+L
CURSOR–UP	CURSOR–UP	CURSOR–UP CTRL+K	CURSOR–UP CTRL+K
CUT	CTRL+X	F10 ESC X	F10 CTRL+ALT+X
DEFAULT–POP–UP	SHIFT+F10	ESC U	SHIFT+F4 CTRL+ALT+U

Key function	Key label		
	Windows graphical interface	UNIX character interface	Windows character interface
DELETE–CHARACTER	DEL	DEL DELETE	DELETE
DELETE–COLUMN	–	ESC CTRL+Z	–
DELETE–END–LINE	–	ESC K	CTRL+ALT+K
DELETE–FIELD	–	ESC CTRL+D	–
DELETE–LINE	–	CTRL+D	CTRL+D
DELETE–WORD	–	ESC D	CTRL+ALT+D
EDITOR–BACKTAB	–	CTRL+B	CTRL+B
EDITOR–TAB	–	CTRL+G TAB	CTRL+G TAB
END	END	END ESC .	END
END–ERROR	ESC	F4 CTRL+E	F4 ESC CTRL+E
ENTER–MENUBAR	ALT	F3 PF3 ESC M	F3 ALT
EXIT	–	ESC Q	CTRL+ALT+Q
FIND	CTRL+F	CTRL+F	CTRL+F
FIND–NEXT	F9	ESC F	CTRL+ALT+F
FIND–PREVIOUS	SHIFT+F9	ESC I	CTRL+ALT+I
GET	F3	F5 ESC O	F5 CTRL+ALT+O
GO	F2	F1 CTRL+X	F1 CTRL+X
GOTO	CTRL+G	ESC G	CTRL+ALT+G

Key function	Key label		
	Windows graphical interface	UNIX character interface	Windows character interface
HELP	F1	ESC ?	–
HOME	HOME	ESC , ESC H	HOME
INSERT–COLUMN	–	ESC CTRL+N	–
INSERT–FIELD	–	ESC CTRL+G	–
INSERT–FIELD–DATA	–	ESC CTRL+F	–
INSERT–FIELD–LABEL	–	ESC CTRL+E	–
INSERT–MODE	INSERT	F9 CTRL+T	INSERT F9 CTRL+T
LEFT–END	HOME	ESC CURSOR–LEFT	ALT+CURSOR–LEFT
MAIN–MENU	–	ESC RETURN ESC CTRL+M	–
MOVE	–	ESC CTRL+V	–
NEW	SHIFT+F3	ESC N	CTRL+ALT+N
NEW–LINE	–	CTRL+N	CTRL+N
NEXT–ERROR	–	ESC E	–
NEXT–FRAME	F6	ESC TAB ESC CTRL+I	–
NEXT–WORD	–	CTRL+W	–
OPEN–LINE–ABOVE	–	ESC L	CTRL+ALT+L
OPTIONS	–	ESC CTRL+O	–
PAGE–DOWN	PAGE–DOWN PGDN NEXT–PAGE NEXT–SCRN	ESC CURSOR– DOWN	PAGE–DOWN

Key function	Key label		
	Windows graphical interface	UNIX character interface	Windows character interface
PAGE–LEFT	–	ESC W	–
PAGE–RIGHT	–	ESC Y	–
PAGE–UP	PAGE–UP PGUP PREV–PAGE PREV–SCRN	ESC CURSOR–UP	PAGE–UP
PASTE	CTRL+V	F12 ESC V	F12 CTRL+ALT+V
PICK	–	ESC CTRL+P	–
PICK–AREA	–	ESC CTRL+W	–
PICK–BOTH	–	ESC CTRL+Q	–
PREV–FRAME	SHIFT+F6	ESC CTRL+U	CTRL+SHIFT+TAB
PREV–WORD	–	CTRL+P	CTRL+P
PUT	F6	F6 ESC S	F6
RECALL	–	F7 CTRL+R	–
REPLACE	–	ESC R	–
REPORTS	–	ESC CTRL+A	–
RESUME–DISPLAY	–	CTRL+Q	–
RETURN	ENTER RETURN CTRL+M	RETURN CTRL+M	ENTER CTRL+M
RIGHT–END	–	ESC CURSOR–RIGHT	ALT+CURSOR–RIGHT
SAVE–AS	SHIFT+F6	ESC A	CTRL+ALT+A
SCROLL–LEFT	–	ESC CTRL+L	–

Key function	Key label		
	Windows graphical interface	UNIX character interface	Windows character interface
SCROLL–MODE	–	ESC T	CTRL+ALT+T
SCROLL–RIGHT	–	ESC CTRL+R	–
SETTINGS	–	ESC CTRL+@	–
STOP	CTRL+BREAK	CTRL+C	CTRL+C
STOP–DISPLAY	–	CTRL+S	–
TAB	TAB	TAB CTRL+I	TAB CTRL+I
TOP–COLUMN	–	ESC CTRL+T	–
UNIX–END	–	CTRL+\	–

Note: If you enter **CTRL+ALT+SHIFT+F1** from a Windows OpenEdge client, graphical or character, OpenEdge displays a window that tells what OpenEdge version you are running.

Changing the function of a key

You can globally change ABL key functions by modifying your environment. For more information on modifying your environment, see the topics on colors and fonts in this book and the topics on user interface environments in *Manage ABL Applications*.

Although ABL has defined functions for several of the keys on your keyboard, you can redefine those keys to perform other functions within an application. In addition, you can assign functions to any of the other keyboard keys.

Suppose the user is accustomed to pressing **F2** to get help information and **F1** to signal that they are finished entering data. Although the AVM usually treats **F2** as `GO` and **F1** as `HELP`, you can switch the functions of those keys as demonstrated in the `i-action.p` procedure.

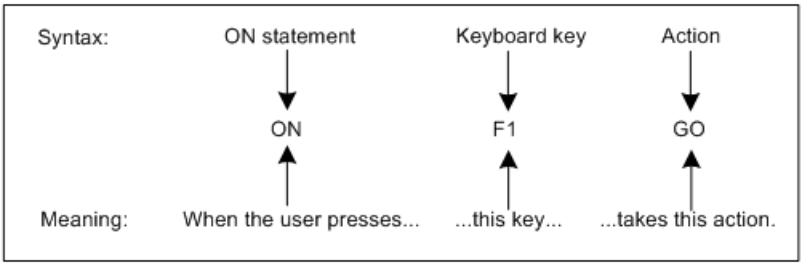
i-action.p

```
ON F1 GO.
ON F2 HELP.
ON CTRL-X BELL.

FOR EACH Customer NO-LOCK:
  DISPLAY Customer.CustNum.
  UPDATE Customer.Name Customer.CreditLlimit Customer.SalesRep.
END.
```

In this procedure, the ON statements redefine the function of **F1**, **F2**, and **CTRL+X**. Run this procedure, then press **F1**. You can see that **F1** now performs the GO function, normally performed by **F2**. If you press **CTRL+X**, the AVM rings the terminal bell. The new key definitions are in effect for the duration of the session, unless you redefine them. Also, on UNIX, any key label you use in an ON statement must have an entry in the PROTERMCAP file for your terminal.

When you use the ON statement, you use the name of the key whose function you are redefining, followed by the action you want to take when the user presses that key, as shown in the following figure.

Figure 21: Key function syntax and meaning



The i-action.p procedure uses three actions: GO, HELP, and BELL. There are many other actions you can use when redefining the function of a key. The following table lists these actions.

Table 38: Actions you assign to keys

Action	Default key
BACKSPACE	**BACKSPACE**
BACK-TAB	**SHIFT+TAB, CTRL+U, CODE+TAB**
BELL	–
CLEAR	**F8, CTRL+Z, CODE+Z**
CURSOR-UP	**×, CTRL+K**
CURSOR-DOWN	**±, CTRL+J**
CURSOR-LEFT	**?, CTRL+H**
CURSOR-RIGHT	**?, CTRL+L**
DELETE-CHARACTER	**DEL**
ENDKEY	–
END-ERROR	**F4, CTRL+E, CODE+E, ESC** (Windows)
ERROR	–
GO	**F2, CTRL+X, CODE+X**
HELP	**F1, HELP, CTRL+W, CODE+W**
HOME	**HOME, ESC H** (UNIX)

Action	Default key
INSERT-MODE	**F3, INSERT, CTRL+T, OVERTYPE**
RECALL	**F7, CTRL+R, CODE+R**
RETURN	**RETURN**
STOP	**CTRL+BREAK** (Windows), **CTRL+C** (UNIX)
TAB	**TAB, CTRL+I**

Using mouse buttons and events

ABL uses a logical (portable) model to reference mouse button input. This ensures that your application works in whatever operating environment it runs. ABL also provides access to the physical mouse input of your operating environment if you need it. Both forms of input are available as events in a manner similar to keyboard events.

Portable and physical buttons

ABL supports four portable mouse buttons:

- **SELECT** — You can select or choose an object by pointing to it and clicking this mouse button. Any previously selected object becomes deselected.
- **EXTEND** — You can toggle the selection state of an object by pointing to it and clicking this button. This has no effect on any other object; therefore, you can use the **EXTEND** button to toggle selection individually on more than one object.
- **MENU** — If an object has an associated pop-up menu, you can activate that menu by clicking this button.
- **MOVE** — By pressing and holding this button you can drag an object on the screen.

Although ABL supports four buttons, the standard mouse used with Windows has only two buttons. Therefore, some physical mouse buttons have double functions, or, you must use control keys with one or more buttons. A Windows mouse can also have three physical buttons. The extra button, called the middle button, is not supported by OpenEdge with Windows for triggers on mouse button events. The following table shows the mappings between the ABL portable mouse buttons and the physical mouse buttons in Windows.

Table 39: Mouse buttons in Windows

Portable button	Windows
SELECT	LEFT mouse button
EXTEND	**CTRL** with LEFT mouse button
MENU	RIGHT mouse button
MOVE	LEFT mouse button

ABL supports two main classes of mouse events—portable and three-button events. You can use portable mouse events to associate triggers with logical actions of any mouse. You can use the three-button mouse events to associate triggers with specific physical actions of a three-button mouse. The names of the portable mouse events come from the mouse key labels listed in the above table (for example, `MOUSE-SELECT-CLICK`). They also correspond to the names of the portable mouse buttons used to generate them. The names of the three-button mouse events correspond to the physical buttons that generate them on a three-button mouse (for example, `LEFT-MOUSE-CLICK`). For a complete description of the names and functions of the portable and three-button mouse events, see *ABL Reference*.

Both portable and three-button mouse events divide into two subclasses—low-level and high-level mouse events. Low-level mouse events are generated by the simplest mouse button actions, while high-level events are generated by more complex actions.

Specifying mouse events

When choosing the events to associate with a trigger, use portable mouse events whenever possible. If you use these events, ABL automatically maps the event to the mouse key the native window system uses to perform the same event. For example, if OpenEdge supports a future window system that uses the right button as the selection button, the `MOUSE-SELECT-CLICK` event will automatically map to the right button for that system, while still mapping to the left button for Windows.

Portable and three-button event priority

If you use three-button mouse events, they take priority over any corresponding portable event. For example, if you define a `MOUSE-SELECT-CLICK` and a `LEFT-MOUSE-CLICK` trigger for the same widget, the AVM only fires the three-button trigger. (In this case, it only fires the `LEFT-MOUSE-CLICK` trigger.)

Note: With Windows, OpenEdge does not support three-button (middle button) events.

Low-level and high-level events

The low-level category consists of those events triggered by mouse button motion in a single direction, such as down or up. The high-level category consists of those events triggered by more complex mouse button motion, such as click or double-click. For ABL, corresponding low-level and high-level mouse events are analogous to equivalent key label and key function events. Like key label and key function events, or three-button and portable mouse events, low-level mouse events take priority over corresponding high-level mouse events. For example, if you define a `MOUSE-SELECT-UP` and `MOUSE-SELECT-CLICK` trigger on the same widget, only the `MOUSE-SELECT-UP` trigger executes.

Like portable events, use high-level mouse events exclusively whenever possible. If you must use low-level events, do not mix low-level and high-level event triggers on the same widget. The processing of both classes of events on the same widget can lead to unintended results.

While the AVM always recognizes corresponding low-level and high-level events, it executes only one trigger for both. (The same is true for corresponding key label and key function events, as well as portable and three-button mouse events). The AVM looks first for a trigger on a low-level event. If there is no trigger on a low-level event, the AVM looks for a trigger on a high-level event. Once it finds and fires a trigger, it waits for the next event. However, each graphic interface may recognize and pass high-level events to the AVM after recognizing a different combination of low-level events. For example, one system might recognize a double-click on a down and another system might recognize the double-click on an up mouse event. This causes mixed event sequences to differ significantly from interface to interface.

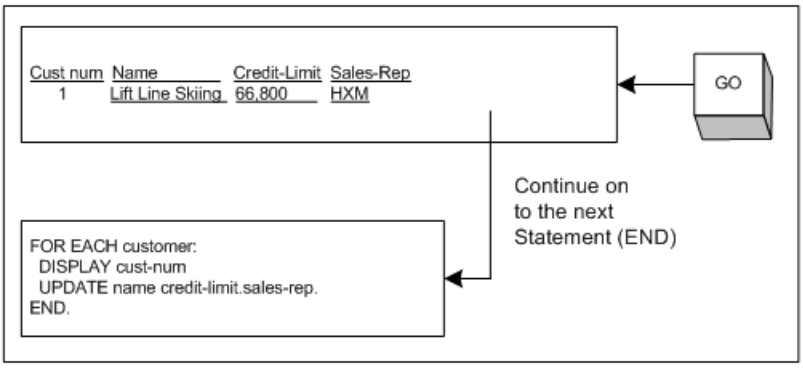
Thus, much like using pixels for frame and window layout, the mixing of low- and high-level mouse events creates portability problems between graphic interfaces. Furthermore, there is no guarantee that the expected event sequences in the same graphic interface might not change in future versions of that interface.

Telling the AVM how to continue processing

When you modify a field or variable, pressing **GO** tells the AVM to accept the data in all the modified fields and variables in the current statement and to go on to the next statement in the procedure.

As the following figure shows, if you press **GO** while the cursor is in the name, credit–limit, or sales–rep fields, the AVM continues on to the next statement in the procedure, which is `END`. The procedure then returns to the beginning of the `FOR EACH` loop.

i-intro.p



You might want to define function keys that tell the AVM to continue processing data a certain way. You can use the `GO-ON` phrase with the `SET` or `UPDATE` statement to do this, as shown in the `i-gon1.p` procedure.

i-gon1.p

```
DISPLAY "You may update each customer." SKIP
  "After making your changes, press one of:" SKIP(1)
  KBLABEL("GO") + "  - Make the change permanent" FORMAT "x(40)" SKIP
  KBLABEL("END-ERROR") + "  - Undo changes and exit" FORMAT "x(40)" SKIP
  "F8  - Undo changes and try again" SKIP
  "F10 - Find next customer" SKIP
  "F12 - Find previous customer"
  WITH CENTERED FRAME ins.

FIND FIRST Customer EXCLUSIVE-LOCK.

upd-loop:
REPEAT:
  UPDATE Customer.CustNum Customer.Name Customer.Address Customer.Address2
    Customer.City Customer.State
    GO-ON(F8 F10 F12) WITH 1 DOWN CENTERED.

  CASE LASTKEY:
    WHEN KEYCODE("F8") THEN
      UNDO upd-loop, RETRY upd-loop.
    WHEN KEYCODE("F10") THEN
      FIND NEXT customer.
    WHEN KEYCODE("F12") THEN
      FIND PREV customer.
  END CASE.
END.
```

In this example, if the user presses **F8**, **F10**, or **F12** while updating the customer data, the procedure immediately goes on to the next statement in the procedure. Let's take a closer look at this procedure.

Any key you can press while running an ABL procedure has a code, a function, and a label associated with it. The code of a key is an integer value that the AVM uses to identify that key. For example, the code of **F1** is 301. The function of a key is the work that the AVM does when you press the key. For example, the function of the **F1** key may be HELP. The label of a key is the actual label that appears on the keyboard key. The label of the **F1** key is F1.

As shown earlier, you can use the KEYLABEL, KEYCODE, KEYFUNCTION, and KBLABEL functions to convert key labels, key codes, and key functions. In addition to these functions, the LASTKEY function returns the key code of the last key pressed.

You can use the functions described in this table to monitor the keys being pressed, as in the i-keys.p procedure.

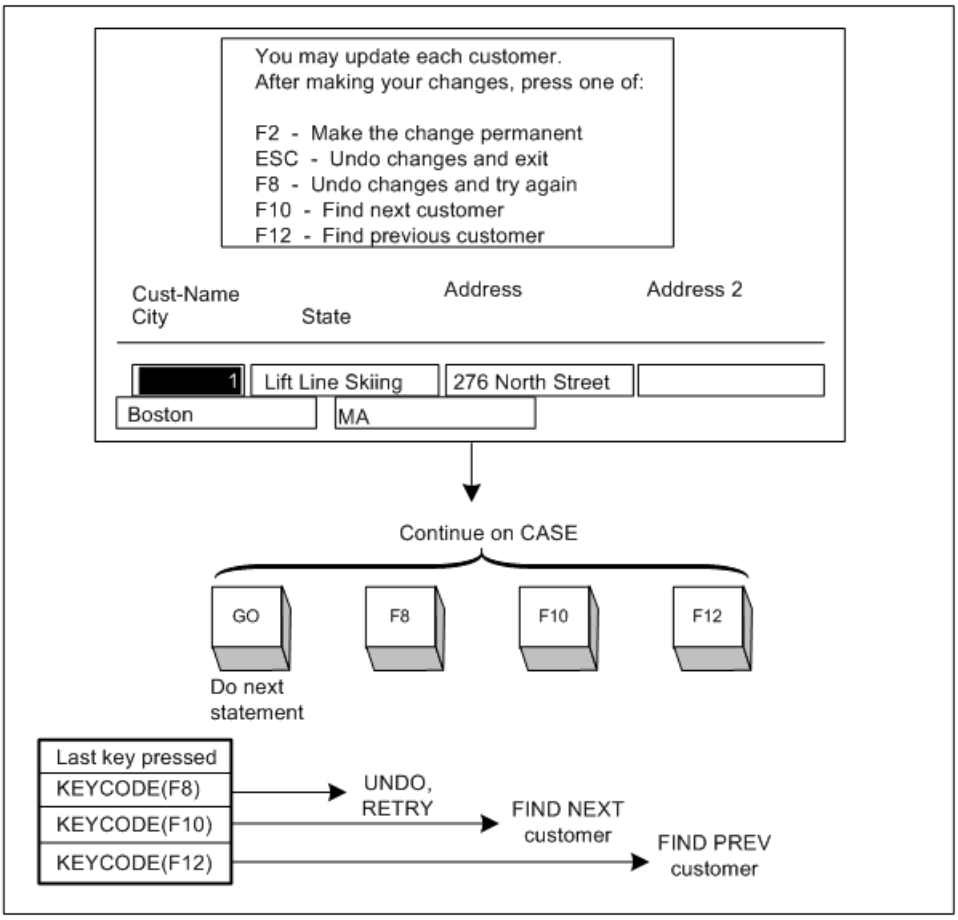
i-keys.p

```
REPEAT:
  DISPLAY "Press any key".
  READKEY.
  DISPLAY
    LASTKEY LABEL "Key Code"
    KEYLABEL(LASTKEY) LABEL "Key Label"
    KEYFUNCTION(LASTKEY) LABEL "Key Function" FORMAT "x(12)".
  IF KEYFUNCTION(LASTKEY) = "end-error" THEN LEAVE.
END.
```

Run procedure i-keys.p to see how the different keys you press translate into key codes, key labels, and key functions.

Now, run the `i-gon1.p` procedure. A screen similar to the one shown in the following figure appears.

Figure 22: The i-gon1.p procedure



While updating the customer information, press **F9**, **F10**, **F11**, or use either of the standard techniques to signal the end of data entry, the AVM goes on to the next statement in the procedure. If you press any other key, the AVM does not continue on to the next statement in the procedure, but instead performs the data entry operation associated with that key. If you press **END-ERROR**, the AVM performs the default ENDKEY processing of `UNDO,` LEAVE.

If the AVM does continue on to the next statement in the procedure, the CASE statement determines the action to take by checking the value of the last key pressed.

The procedure `i-gon2.p` shows how you can achieve the same functionality in an event-driven application.

i-gon2.p

```
FORM
  Customer.CustNum Customer.Name Customer.Address Customer.Address2
  Customer.City Customer.State
  WITH CENTERED FRAME upd-frame.

DISPLAY "You may update each customer." SKIP
  "After making your changes, press one of:" SKIP(1)
  KBLABEL("GO") + "  - Make the change permanent" FORMAT "x(40)" SKIP
  KBLABEL("END-ERROR") + "  - Undo changes and exit" FORMAT "x(40)" SKIP
  "F8  - Undo changes and try again" SKIP
  "F10 - Find next customer" SKIP
  "F12 - Find previous customer"
  WITH CENTERED FRAME ins.

ON F8 ANYWHERE DO:
  DISPLAY Customer.CustNum Customer.Name Customer.Address Customer.City
     Customer.State
    WITH CENTERED FRAME upd-frame.
END.

ON F10 ANYWHERE DO:
  APPLY "GO" TO FRAME upd-frame.
  FIND NEXT Customer.
  DISPLAY Customer.CustNum Customer.Name Customer.Address Customer.Address2
    Customer.City Customer.State
    WITH CENTERED FRAME upd-frame.
END.

ON F12 ANYWHERE DO:
  APPLY "GO" TO FRAME upd-frame.
  FIND PREV Customer.
  DISPLAY Customer.CustNum Customer.Name Customer.Address Customer.Address2
    Customer.City Customer.State
    WITH CENTERED FRAME upd-frame.
END.

ON GO OF FRAME upd-frame
  ASSIGN FRAME upd-frame Customer.CustNum Customer.Name Customer.Address
    Customer.Address2 Customer.City Customer.State.

ENABLE ALL WITH FRAME upd-frame.
FIND FIRST customer.
APPLY "F8" TO FRAME upd-frame.

WAIT-FOR END-ERROR OF FRAME upd-frame.
```

Use the ANYWHERE option of the ON statement to set up triggers that execute no matter which widget has input focus. In i-gon2.p, the ANYWHERE option is used to assign triggers to function keys.

Note: In a larger program, you must be careful of the scope of the ANYWHERE trigger.

Monitoring keystrokes during data entry

ABL provides two methods to monitor keystrokes:

- Editing blocks on page 290
- User interface triggers on page 290

Editing blocks

Prior to Progress Version 7 and user interface triggers, editing blocks were the only method available to monitor individual keystrokes. An editing block is part of an UPDATE, SET, or PROMPT–FOR statement that allows the programmer to read and process each keystroke individually. Editing blocks are used for the following purposes:

- To enable a few special keys in one or more fields while allowing normal data entry
- To disallow normal data entry in one or more fields and instead allow the use of only a few special keys to change the value

Each type of editing block can now be replaced with user interface triggers.

User interface triggers

ABL allows you to access all keystrokes as events. You can intercept these events using the ON statement or the trigger phrase of any statement that declares a user interface widget. For each intercepted event, you can provide a user interface trigger to implement any ABL action. This action can add to, and often replace, the default action associated with the event.

All ABL key labels and key functions are valid events. You can specify a keyboard event in the ON statement or trigger phrase by:

- Using the key label name, such as **F2**
- Using the key function name, such as GO

Key label events take priority over corresponding key function events. For example, if you specify a trigger for the TAB key function event and another trigger on the same widget for the **CTRL+I** key label event, only the **CTRL+I** trigger executes.

Where a key label event corresponds to a key function, use the key function event whenever possible. Key functions are more portable. If you must reference key label events, be sure not to also reference corresponding key function events for the same widget.

Using editing blocks and user interface triggers

Note: EDITING blocks are a deprecated feature. Prior to event-driven programming, ChUI procedural applications used EDITING blocks. EDITING blocks are not appropriate for event-driven, multi-tier, or Service-Oriented applications. Use the WAIT-FOR statement to replace EDITING blocks.

The i-kystka.p procedure uses an EDITING block on the UPDATE statement to enable a special key while allowing normal data entry.

i-kystka.p

```
DEFINE FRAME cust-frame
  Customer.CustNum SKIP Customer.Name SKIP Customer.Address SKIP
  Customer.Address2 SKIP Customer.City Customer.State SKIP
  Customer.SalesRep HELP "To see a list of values, press F6."
  WITH DOWN SIDE-LABELS.

REPEAT WITH FRAME cust-frame:
  PROMPT-FOR Customer.CustNum.
  FIND Customer USING Customer.CustNum.
  UPDATE Customer.Name Customer.Address Customer.Address2 Customer.City
    Customer.State Customer.SalesRep EDITING:
    READKEY.
    IF FRAME-FIELD = "salesrep" AND LASTKEY = KEYCODE("F6") THEN DO:
      FOR EACH SalesRep NO-LOCK:
        DISPLAY SalesRep.SalesRep
          WITH NO-LABELS 9 DOWN COLUMN 60 ROW 5.
      END.
    END.
    ELSE
      APPLY LASTKEY.
  END. /* UPDATE */
END. /* REPEAT */
```

The EDITING block processes every keystroke the user enters while that UPDATE statement is executing. Within the EDITING block, the READKEY statement reads a keystroke from the user. The associated key code is automatically stored as LASTKEY. The procedure uses LASTKEY and the FRAME-FIELD function to determine whether the key is **F6**, and whether it was entered from within the SalesRep field. If so, it displays a list of all known sales reps. If the key is not **F6**, or the current field is not the SalesRep field, then the APPLY statement is executed. This causes the AVM to handle the keystroke normally.

The i-kystkb.p procedure uses a user interface trigger in place of the EDITING block to achieve the same functionality.

i-kystkb.p

```
DEFINE FRAME cust-frame
  Customer.CustNum SKIP Customer.Name SKIP Customer.Address SKIP
  Customer.Address2 SKIP Customer.City Customer.State SKIP
  Customer.SalesRep HELP "To see a list of values, press F6."
  WITH DOWN SIDE-LABELS.

ON F6 OF Customer.SalesRep DO:
  FOR EACH SalesRep NO-LOCK:
    DISPLAY SalesRep.SalesRep
      WITH NO-LABELS 9 DOWN COLUMN 60 ROW 5.
  END.
END.

REPEAT WITH FRAME cust-frame:
  PROMPT-FOR Customer.CustNum.
  FIND Customer EXCLUSIVE-LOCK USING Customer.CustNum.
  UPDATE Customer.Name Customer.Address Customer.Address2 Customer.City
    Customer.State Customer.SalesRep.
END.
```

Use a trigger in place of the EDITING block to produce simpler, cleaner code. It would be easier to rewrite this code to be more event driven. First, replace the UPDATE statement with ENABLE, WAIT-FOR, and ASSIGN. You then might remove the REPEAT block and define a button that the user can select to move to the next **Customer** record. You might also replace the display of sales reps with a selection list or browse widget.

The i-kystkbk.p procedure uses an EDITING block to disallow normal data entry on a field and allow only a special key.

i-kystk.p

```
DEFINE FRAME cust-frame
  Customer.CustNum SKIP Customer.Name SKIP Customer.Address SKIP
  Customer.Address2 SKIP Customer.City Customer.State SKIP
  Customer.SalesRep HELP "Use the space bar to scroll the values."
  WITH SIDE-LABELS.

REPEAT WITH FRAME cust-frame:
  PROMPT-FOR Customer.Custnum.
  FIND Customer EXCLUSIVE-LOCK USING Customer.Custnum.
  UPDATE Customer.Name Customer.Address Customer.City Customer.State
     Customer.SalesRep EDITING:
    READKEY.
    IF FRAME-FIELD <> "salesrep" THEN
      APPLY LASTKEY.
    ELSE DO:
      IF LASTKEY = KEYCODE(" ") THEN DO:
        FIND NEXT SalesRep NO-LOCK NO-ERROR.
        IF NOT AVAILABLE SalesRep THEN
          FIND FIRST SalesRep NO-LOCK.
        DISPLAY SalesRep.SalesRep @ Customer.SalesRep.
      END.
      ELSE IF LOOKUP(KEYFUNCTION(LASTKEY),
        "TAB,BACK-TAB,GO,END-ERROR") > 0 THEN APPLY LASTKEY.
      ELSE BELL.
    END.
  END. /* UPDATE */
END. /* REPEAT */
```

Like i-kystka.p, this procedure uses an EDITING block on the UPDATE statement. In the EDITING block, it uses READKEY, LASTKEY, and FRAME-FIELD to obtain and analyze a keystroke. If the keystroke is not in the Sales-rep field, it is processed normally. Within the SaleRep field, only the spacebar is treated specially and only the TAB, BACK-TAB, GO, and END-ERROR key functions are treated normally. If the user types any other key within the field, the terminal bell sounds. When the user presses the spacebar in the SalesRep field, the value of that field and the SalesRegion field change.

The i-kystk2.p procedure uses triggers to accomplish the same thing.

i-kystk2.p

```
DEFINE FRAME cust-frame
  Customer.Custnum SKIP Customer.Name SKIP Customer.Address SKIP
  Customer.Address2 SKIP Customer.City SKIP Customer.State SKIP
  Customer.SalesRep HELP "Use the space bar to scroll the values"
  WITH SIDE-LABELS.

ON " " OF Customer.SalesRep DO:
  FIND NEXT SalesRep NO-LOCK NO-ERROR.
  IF NOT AVAILABLE SalesRep THEN
    FIND FIRST SalesRep NO-LOCK.
  DISPLAY SalesRep.SalesRep @ Customer.SalesRep WITH FRAME cust-frame.
  RETURN NO-APPLY.
END.

ON ANY-PRINTABLE OF Customer.SalesRep DO:
  BELL.
  RETURN NO-APPLY.
END.

REPEAT WITH FRAME cust-frame:
  PROMPT-FOR Customer.CustNum.
  FIND Customer EXCLUSIVE-LOCK USING Customer.CustNum.
  FIND SalesRep OF Customer NO-LOCK.
  UPDATE Customer.Name Customer.Address Customer.Address2 Customer.City
    Customer.State Customer.SalesRep.
END.
```

Note the use of `RETURN NO-APPLY` in both the `ANY-PRINTABLE` and spacebar trigger. This is equivalent to omitting the `APPLY LASTKEY` statement in an `EDITING` block. Thus, the spacebar trigger brings the next `SalesRep` into view, without also inserting a space character in the field.

Note also that in `i-kystk2.p` the `ANY-PRINTABLE` trigger rejects all keys that enter data characters other than a space. This works together with the spacebar trigger to allow only the spacebar and navigation keys (`TAB`, etc.) in the field.

Sometimes an `EDITING` block defines a special key that applies for all active fields, as shown in the `i-kystk3.p` procedure.

i-kystk3.p

```
DEFINE FRAME cust-frame
  Customer.CustNum SKIP Customer.Name SKIP Customer.Address SKIP
  Customer.Address2 SKIP Customer.City Customer.State SKIP
  Customer.SalesRep WITH SIDE-LABELS.

REPEAT WITH FRAME cust-frame:
  PROMPT-FOR Customer.CustNum.
  FIND Customer EXCLUSIVE-LOCK USING Customer.CustNum.
  MESSAGE "Press F6 to see the previous Customer.".
  UPDATE Customer.Name Customer.Address Customer.Address2 Customer.City
    Customer.State Customer.SalesRep EDITING:
    READKEY.
    IF KEYLABEL(LASTKEY) = "F6" THEN DO:
      FIND PREV Customer NO-LOCK NO-ERROR.
      IF NOT AVAILABLE Customer THEN
        FIND FIRST Customer NO-LOCK.
      DISPLAY Customer.CustNum Customer.Name Customer.Address Customer.City
        Customer.State Customer.SalesRep
        WITH FRAME cust-frame.
    END.
    ELSE APPLY LASTKEY.
  END.
  HIDE MESSAGE.
END.
```

In `i-kystk3.p`, the `EDITING` block defines a special key, **F6**, that is available in all fields of the `UPDATE` statement. When the user presses this key, the previous **Customer** record displays.

The `i-kystk4.p` procedure uses a trigger to achieve the same result.

i-kystk4.p

```
DEFINE FRAME cust-frame
  Customer.CustCum SKIP Customer.Name SKIP Customer.Address SKIP
  Customer.Address2 SKIP Customer.City Customer.State SKIP
  Customer.SalesRep WITH SIDE-LABELS.

ON F6 ANYWHERE DO:
  IF FRAME-FIELD = "custnum" THEN
    RETURN NO-APPLY.

  FIND PREV Customer NO-ERROR.
  IF NOT AVAILABLE Customer THEN
    FIND FIRST Customer.

  DISPLAY Customer.CustNum Customer.Name Customer.Address Customer.City
    Customer.State Customer.Salesep
    WITH FRAME cust-frame.
END. /* ON F6 */

REPEAT WITH FRAME cust-frame:
  PROMPT-FOR Customer.CustNum.
  FIND Customer EXCLUSIVE-LOCK USING Customer.CustNum.
  MESSAGE "Press F6 to see the previous Customer.".
  UPDATE Customer.Name Customer.Address Customer.Address2 Customer.City
    Customer.State Customer.SalesRep.
  HIDE MESSAGE.
END.
```

If you define a trigger to be active ANYWHERE, then it applies to all widgets. In i-kystk4.p, the F6 trigger executes whenever the user presses **F6** while input is enabled. Within the trigger, the FRAME-FIELD function determines whether the t rigger executes from the UPDATE or PROMPT-FOR statement. If it is the PROMPT-FOR statement, then **F6** is ignored; if it is the UPDATE statement, the previous **Customer** record displays.

In a larger program, be careful of the scope of the ANYWHERE trigger. You are usually better off listing the specific widgets to which a trigger applies. For example, you could rewrite the ON statement in i-kystrk4.p as follows:

```
ON F6 OF Name, Address, Address2, City, State, SalesRep
```

If you take this approach, you can remove the code that checks whether FRAME-FIELD is CustNum within the trigger.

7

Alternate I/O Sources

Most of the procedures you have seen so far have used the terminal as the input source and as the output destination.

However, you've probably already thought of situations in which you might want to get data from and send data to locations other than the terminal. For example, you might want to send reports to your printer or an operating system file.

For details, see the following topics:

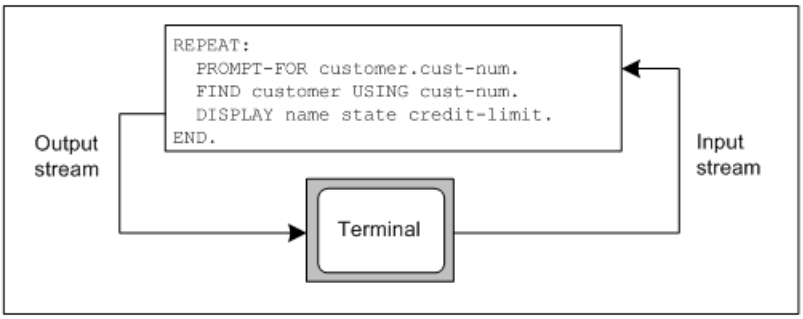
- Understanding input and output
- Changing the output destination
- Changing the input source of a procedure
- Defining additional input/output streams
- Sharing streams among procedures
- Summary of opening and closing streams
- Processes as input and output streams (NT and UNIX only)
- I/O redirection for batch jobs
- Reading the contents of a directory
- Performing code-page conversions
- Converting nonstandard input files

Understanding input and output

When an ABL (Advanced Business Language) procedure gets input from the terminal, it uses an *input stream*. Similarly, when the procedure sends output to the terminal, it uses an *output stream*.

Every procedure automatically gets one input stream and one output stream. These are called the *unnamed streams*. By default, ABL assigns both of these unnamed streams to the terminal, as shown in the following figure.

Figure 23: The unnamed streams



This procedure contains no special syntax about where to get and send data so ABL automatically assigns the input stream and the output stream to the terminal.

Changing the output destination

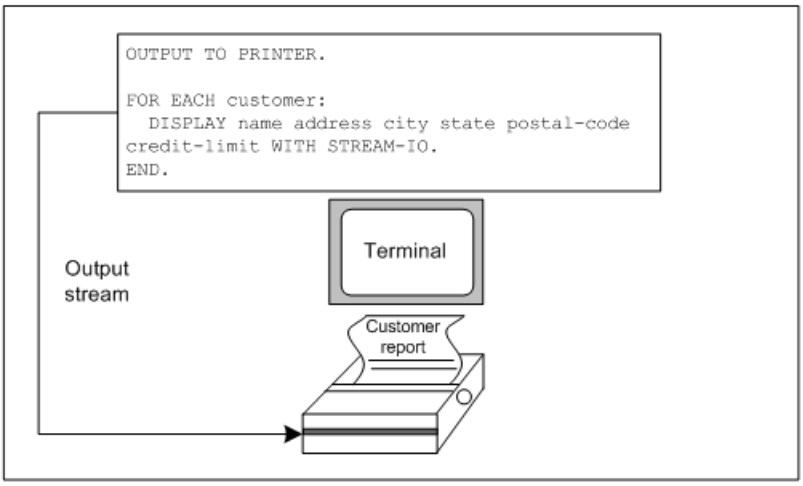
You use the `OUTPUT TO` statement to name a new destination for output. All statements that output data (such as `DISPLAY` or `SET`) use the new output destination. The possible destinations include:

- The printer
- An operating system file or device
- The terminal (by default)
- The system clipboard (Windows only)

Output to printers

The following figure shows the output stream being redirected to a printer using the PRINTER option of the OUTPUT TO statement.

Figure 24: Redirecting the unnamed output stream



If you run the procedure, you do not see anything displayed on your terminal because all the output from the DISPLAY statement goes to the printer. The STREAM-IO Frame phrase option formats the output especially for character-based destinations. See Stream I/O vs. screen I/O on page 303 for more information.

Additional options for printing in Windows

In Windows, you can also get a list of printers, change the default printer for the session, and provide access to the **Print** dialog box.

Getting a list of printers

You can use the GET-PRINTERS option of the SESSION statement to obtain a list of the printers configured for the current system. The i-wgetls.p procedure shows you how to obtain the list of currently configured printers, select a printer, and print to it:

i-wgetls.p

```
/*1*/ DEFINE VARIABLE printer-list AS CHARACTER NO-UNDO.
      DEFINE VARIABLE printer-idx  AS INTEGER   NO-UNDO.

/*2*/ printer-list = SESSION:GET-PRINTERS().
/*3*/ printer-idx  = LOOKUP("SpecialPrinter", printer-list).

/*4*/ IF printer-idx > 0 THEN DO:
        MESSAGE "output" VIEW-AS ALERT-BOX.
        OUTPUT TO PRINTER "SpecialPrinter".
      END.
/*5*/ ELSE
        OUTPUT TO PRINTER.

      FOR EACH Customer NO-LOCK
        WHERE Customer.Country = "Finland" :
        DISPLAY Customer.Name Customer.Country.
      END.

/*6*/ OUTPUT CLOSE.
```

The following list explains the important elements in `i-wgetls.p`:

1. Create two variables: one for the output of `SESSION:GET-PRINTERS`; the other, for the output of `LOOKUP`.
2. The `SESSION:GET-PRINTERS` method returns a comma-separated list of printers that are currently configured on the system.
3. The `LOOKUP` function obtains an integer that gives the position of "SpecialPrinter" in the list. If the printer is not in the list, it returns a 0.
4. The `IF` statement determines whether the printer "SpecialPrinter" is available and if so, prints to it.
5. If the printer "SpecialPrinter" is not available, the `ELSE DO` statement prints to the default printer.
6. The `OUTPUT CLOSE` statement stops sending output to the current destination and redirects output to the destination used prior to `OUTPUT TO`. See Stream I/O vs. screen I/O on page 303 for more information.

Changing the default printer

To change the default printer for the session, you can use the `PRINTER-NAME` option of the `SESSION` statement as shown in the `i-wchpr.p` procedure.

i-wchpr.p

```
/*1*/ DEFINE VARIABLE printername AS CHARACTER NO-UNDO.

/*2*/ printername          = SESSION:PRINTER-NAME.
/*3*/ SESSION:PRINTER-NAME = "\\AB1\hplaser".
/*4*/ OUTPUT TO PRINTER.

      FOR EACH Customer NO-LOCK:
        DISPLAY Customer.Name SPACE Customer.Balance.
      END.

/*5*/ SESSION:PRINTER-NAME = printername.
```

To change the default printer:

1. Create a variable for the output of SESSION:PRINTER-NAME.
2. The SESSION:PRINTER-NAME attribute returns the name of the default printer.
3. The SESSION:PRINTER-NAME attribute sets another printer, \\AB1\hplaser, as the default printer.
4. The OUTPUT TO PRINTER prints the report on the printer \\AB1\hplaser.
5. The SESSION:PRINTER-NAME attribute restores the original default printer.

Providing access to the print dialog box

You can use the SYSTEM-DIALOG PRINTER-SETUP statement to provide access to the Windows print dialog box. This allows the application user to set up the printer or even change the printer that receives the output.

Output to files or devices

You can direct the report to a standard text file. Replace the PRINTER option of the OUTPUT TO statement in the Figure 24: Redirecting the unnamed output stream on page 299 figure with the following code:

```
OUTPUT TO rpt-out PAGED.
```

In this OUTPUT TO statement, rpt-out is the name of the file where you direct the output. ON UNIX, the filename is case sensitive. That is, UNIX treats rpt-out and RPT-OUT as two different files. However, to most other operating systems, these are the same file.

The PAGED option indicates that you want a page break in the output every 56 lines. PAGED is automatic for output to a printer. If you do not use the PAGED option, ABL sends the data to the file continuously without any page break control characters.

You can also use the VALUE option to specify filenames stored in variables or fields. For information, see Sending output to multiple destinations on page 302.

Some operating systems (like UNIX) allow you to send output to devices other than printers, terminals, and disk files using a name (like a filename) to reference them. You redirect output to these devices exactly like files.

Output to the clipboard

In general, you use the CLIPBOARD system handle to write output to the system clipboard (in graphical environments) or OpenEdge clipboard (in character environments). However, in Windows, you can use the OUTPUT TO statement to redirect output to the system clipboard using the "CLIPBOARD" option (quotes required). This allows you to use ABL output statements (such as DISPLAY) to buffer up to 64K of data to the clipboard. You send the buffered data to the clipboard by changing or closing the output destination (using the OUTPUT CLOSE statement). For more information on changing and closing the output destination, see Sending output to multiple destinations on page 302. For more information on using the CLIPBOARD system handle for output (and also input), see System Clipboard on page 437.

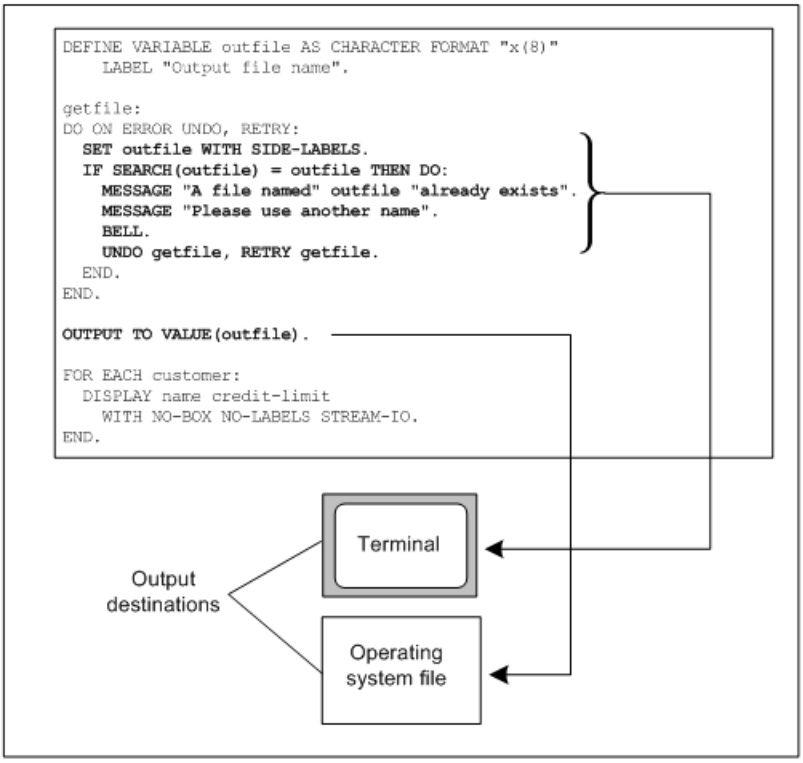
Resetting output to the terminal

You can reset any output destination to the terminal using the TERMINAL option of the OUTPUT TO statement. For more information, see Sending output to multiple destinations on page 302.

Sending output to multiple destinations

At some points in a procedure, you might want to send output to the terminal, but at other points you might want to send output to a file. ABL does not restrict you to one output destination per procedure.

Figure 25: Multiple output destinations



In the procedure in the above figure, you supply the name of the file where you want to send a customer report and then, if that file does not already exist, send the report to that file.

To send output to a file:

1. The SET statement prompts you for the filename.
2. The SEARCH function searches for the file, returning the filename if the file is found.
3. If the file is found, the procedure:
   - Displays a message telling you the file already exists and to use another filename
   - Rings the terminal bell
   - Undoes the work done in the DO block and retries the block, giving you the opportunity to supply a different filename
4. If the file is not found, the procedure uses this statement:

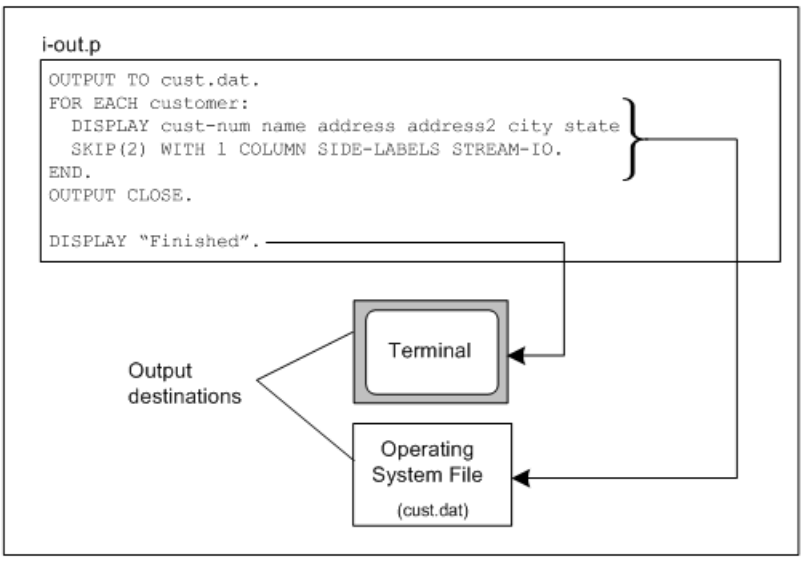
```
OUTPUT TO VALUE(outfile).
```

   This statement redirects the output to the file you specified. You must use the VALUE keyword with the OUTPUT TO statement. The VALUE option tells the AVM (ABL Virtual Machine) to use the value of the outfile variable rather than the name "outfile" itself. If instead you say OUTPUT TO outfile, the AVM assumes that outfile is the name of the text file to which you want to send output.

You use the OUTPUT CLOSE statement to stop sending output to a destination. Output sent after the OUTPUT CLOSE statement goes to the destination used prior to the OUTPUT TO statement.

For example, the procedure in the following figure uses the OUTPUT CLOSE statement to reset the output destination from a file to the terminal.

Figure 26: The OUTPUT CLOSE statement



The procedure in the above figure sends customer information to a file called cust.dat. Then the procedure displays the word "Finished" on your terminal screen. The procedure executes as follows:

1. The OUTPUT TO statement redirects output so all statements that normally send output to the terminal send output to the cust.dat file.
2. The FOR EACH customer and DISPLAY statements produce a report listing each customer's name, address, city, and state. The procedure sends the report to the cust.dat file.
3. The OUTPUT CLOSE statement resets the output destination for the procedure from the cust.dat file to the terminal.
4. The last DISPLAY statement displays the message "Finished" on the terminal screen.

Stream I/O vs. screen I/O

When you compile a procedure, ABL automatically lays out the frames as appropriate for the current screen display. The layout differs across user interfaces because some fields have special decoration. The font also affects the size, and therefore the layout, of fields.

However, when you are displaying a frame to a file or printer rather than the screen, you do not want ABL to design the frame for screen display. You want ABL to lay out the field for a character display using a fixed font and fields without decorations. Otherwise, output written to a printer or operating system file may be unattractive or even partially unreadable. You also want ABL to represent all data fields as text, not as graphical widgets such as editors or sliders.

ABL provides two mechanisms to deal with these issues:

- The STREAM-IO option of the COMPILE statement
- The STREAM-IO and SCREEN-IO options of the Frame phrase

If all output from a procedure is to printers or operating system files, you can use the STREAM-IO option of the COMPILE statement. This forces ABL to lay out all frames using a standard fixed font and to display all data fields as simple text fields. The following widget types are converted to text:

- Editors
- Selection lists
- Sliders
- Radio sets
- Toggle boxes

If you only write a few frames to a printer or file in your procedure, you can use the STREAM-IO frame option to mark those frames. This forces ABL to lay out only those specific frames for output to a printer or file. If you do not specify STREAM-IO in the COMPILE statement, then all other frames are designed for screen display.

If only a few screens in your procedure are displayed to a screen, you use the SCREEN-IO frame option to mark those frames. Then compile with STREAM-IO so that all other frames are laid out for display to a printer or file.

A printing solution

The OpenEdge ADE toolset provides a portable solution for printing text files. The solution is a procedure called _osprint.p and it is located in the adecomm procedure library in the OpenEdge product directory (DLC). You can also access the source code for this procedure in the src/adecomm.pl file located in the OpenEdge product directory.

The _osprint.p procedure sends a specified text file to the default printer as paged output. Input parameters for the procedure allow you to specify values that configure a print job. In Windows, you can also direct the _osprint.p procedure to display the **Print** dialog box and print the text in a specified font. Use the following syntax to call the _osprint.p procedure from an ABL procedure:

Syntax

```
RUN adecomm/_osprint.p
  ( INPUT parentWindow , INPUT printFile ,
    INPUT fontNumber , INPUT PrintFlags . INPUT pageSize ,
    INPUT pageCount , OUTPUT result ).
```

The parameters of the _osprint.p procedure are as follows:

INPUT *parentWindow*

A window handle identifying the parent window for **Print** dialog box and any print status messages in Windows. The procedure ignores a value specified for this parameter in character interfaces. If you specify the Unknown value (?) or an invalid handle in Windows, the procedure uses the CURRENT-WINDOW handle.

INPUT *printFile*

A string value representing the name of a text file to print. You can specify an absolute or relative path for the file. The _osprint.p procedure uses the PROPATH to locate the file.

INPUT *fontNumber*

An integer value representing an entry in the font table maintained by the FONT-TABLE handle. The _osprint.p procedure uses the specified font to print the text file in Windows. The procedure ignores a value specified for this parameter in character interfaces. If you specify the Unknown value (?) or an integer value that does not exist in the font table for Windows, the procedure uses the default system font to print the text file.

INPUT *PrintFlags*

An integer value that determines which printing options are used for a print job in Windows (only). You can use the values in the following table. If you need to use more than one option, add the values of the options together. In all cases, the _osprint.p procedure sets the value of the PRINTER-CONTROL-HANDLE attribute of the SESSION handle to zero (0).

Table 40: Printing options for Windows

Printing option	Value	Selection
Context	0	Default print context
	1	**Print** dialog box. This allows the user to establish a print context
Landscape orientation	2	Landscape orientation
Paper Size	4	Letter
	8	Legal
	12	A4
Paper Tray	32	Upper tray
	64	Middle tray
	96	Lower tray
	128	Manual
	160	Auto
UTF-8	512	Required when sending UTF-8 encoded data to the printer

INPUT *pageSize*

An integer value representing the number of lines per page. If you specify zero (0) for this parameter, the printer determines the page size. Windows ignores this parameter and calculates the page size based on the **Paper Size** setting in the **Print Setup** dialog box and the font specified with the *fontNumber* parameter.

Note: The maximum number of character per line is 255.

INPUT *pageCount*

An integer value that determines if _osprint.p prints the entire text file or a range of pages from the text file in Windows. The procedure ignores a value specified for this parameter in character interfaces. If the value of this parameter is not zero (0) in Windows, ABL uses the page range specified for the current print context.

OUTPUT result

A logical value that reports the success or failure of the print job.

To call the _osprint.p procedure from an ABL procedure, you must define a variable for the result output parameter. The _osprint.p procedure is an example.

_osprint.p

```
/*1*/ DEFINE VARIABLE result AS LOGICAL NO-UNDO.

/*2*/ OUTPUT TO tmp.dat.
      FOR EACH Customer NO-LOCK:
        DISPLAY Customer.CustNum Customer.Name Customer.Phone WITH STREAM-IO.
      END.
      OUTPUT CLOSE.

/*3*/ RUN adecomm/_osprint.p (INPUT ?, INPUT "tmp.dat", INPUT ?, INPUT 1,
                              INPUT 0, INPUT 0, OUTPUT result).
/*4*/ OS-DELETE "tmp.p".
```

The following list describes the important elements of the _osprint.p procedure:

1. Create a variable for the OUTPUT parameter of the _osprint.p procedure.
2. Generate the temporary text file to be printed. Remember to close the output stream after generating the text file.
3. Run the _osprint.p procedure to print the generated text file.
4. Delete the temporary text file.

For more information on the language elements referenced in this section, see *ABL Reference*.

Changing the input source of a procedure

You use the INPUT FROM statement to name a new source for input. The possible sources include an operating system file or device, the contents of an operating system directory, or the terminal (by default).

All statements (such as UPDATE or SET) that require data use the new input source.

Input from files

A data file (not a database table) that contains information on new customers might look like i-datf1.d.

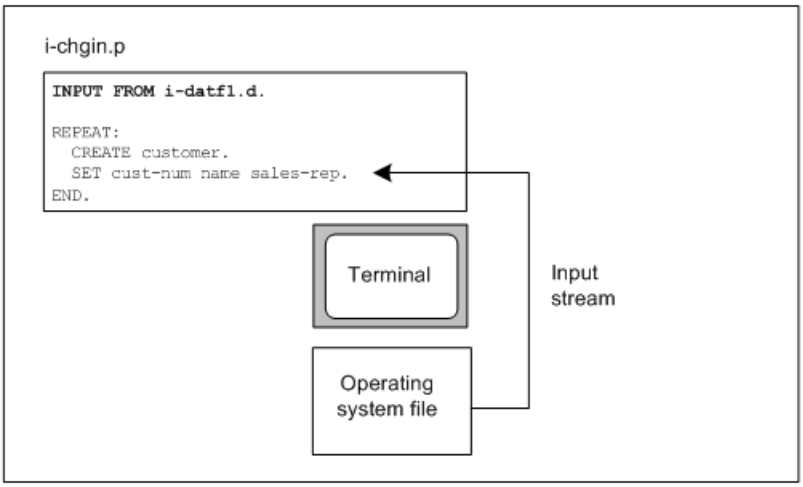
i-datf1.d

```
90 "Wind Chill Hockey" BBB
91 "Low Key Checkers"  DKP
92 "Bing's Ping Pong"  SLS
```

This file is in standard ABL format. That is, blanks separate the field values. Field values that contain embedded blanks are surrounded by quotation marks (" "). See Converting nonstandard input files on page 318 for information on using alternate formats.

You can write an ABL procedure and tell that procedure to get its input from the `i-datf1.d` file, as shown in the following figure.

Figure 27: Redirecting the unnamed input stream



The SET statement, which normally gets its input from the terminal, gets its input from the `i-datf1.d` file. The cust–num field uses the first data item, 90. The name field uses the next quoted data item, "Wind Chill Hockey," etc. Each time ABL processes a data entry statement, it reads one line from the file.

Note: For information about the size limit of input files, see the input/output limits section of *Manage ABL Applications*.

Input from devices, directories, and the terminal

Some operating systems (like UNIX) let you receive input from devices other than terminals and disk files and let you use a name (like a filename) to reference them. You redirect input from these devices exactly like files. For more information, see Input from files on page 306.

You can read the contents of a directory using the `OS-DIR` option of the `INPUT FROM` statement. For more information on reading the contents of a directory, see Reading the contents of a directory on page 315.

You can reset any input source to the terminal using the `TERMINAL` option of the `INPUT FROM` statement. For more information, see Receiving input from multiple sources on page 307.

Receiving input from multiple sources

At some points in a procedure you might want to get input from the terminal, but at other points you might want to get input from a file. A single procedure can use multiple input sources.

For example, suppose you want to create records for the customers in the `i-datf1.d` data file. Before creating the records, you probably want to display the customer numbers in the file and ask if the user wants to create customer records for those numbers. To do this, you need input from the terminal. If the user wants to create customer records for the customers in the `i-datf1.d` file, you also need input from the file.

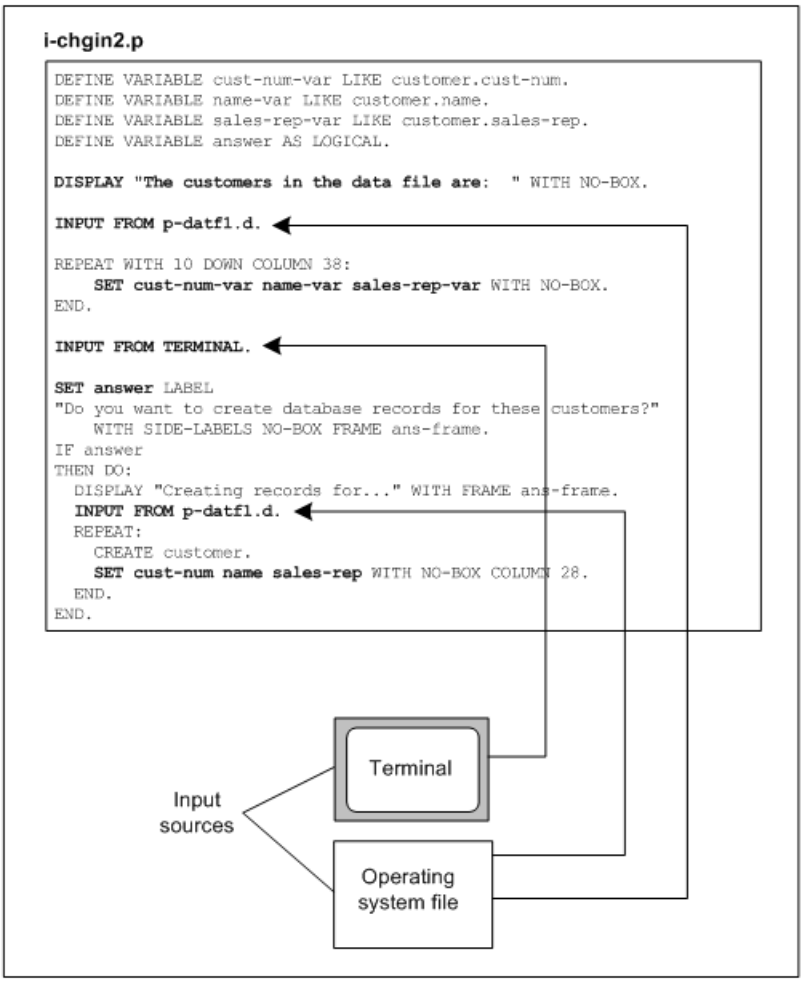
The i-chgin2.p procedure uses multiple input sources to perform the work described above. Because i-chgin2.p uses the same data file (i-datf1.d) you used in the previous section to create customer records, you must delete customers 90, 91, and 92 from your database before you run i-chgin2.p. Use the i-io3.p procedure to delete the customers.

i-io3.p

```
FOR EACH Customer WHERE Customer.CustNum > 89:
  DELETE Customer.
END.
```

The following figure shows the i-chgin2.p procedure.

Figure 28: Multiple input sources



These are the specific steps that the i-chgin2.p procedure follows:

1. The DISPLAY statement displays some text.
2. The first INPUT FROM statement redirects the input source to the i-datf1.d file.
3. The SET statement assigns the values in the i-datf1.d file to the cust-num-var, name-var, and sales-rep-var variables. As it assigns the values to these variables, you see the values on the terminal screen.
4. The INPUT FROM TERMINAL statement redirects the input source to the terminal. The INPUT CLOSE statement could have been used instead of the INPUT FROM TERMINAL statement. However, since this

procedure might have been called from another procedure, it is better to be explicit about the input source you want to use.

5. The SET answer statement prompts you to create database records for the customer data just displayed. If you answer yes, the procedure:
   - Redirects the input source to come from the beginning of the `i-datf1.d` file
   - Creates a customer record and assigns values to the cust–num., name, and sales–rep fields in that record for each iteration of a REPEAT block
   - Ends the REPEAT block when the SET statement reaches the end of the input file

Reading input with control characters

You can use the BINARY option of the `INPUT FROM` statement to read control characters. Input sources often contain control characters that affect the format or quantity of data that you receive. For example, a text file might contain NUL ("\0") characters to terminate character strings. Without the BINARY option, ABL ignores all input on a line after the first NUL character. The BINARY option allows you to read all the data in the file, including any NUL and other non-printable control characters without interpretation.

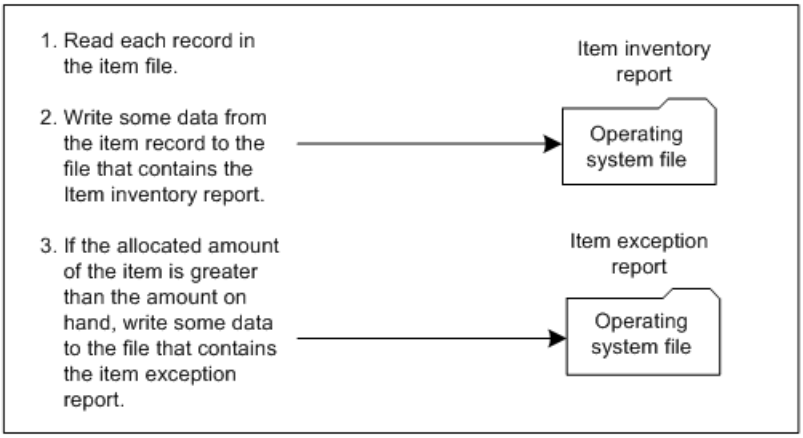
Defining additional input/output streams

When you start a procedure, the AVM automatically provides that procedure with input and output streams. As described in the previous sections, the default source for the input stream is the terminal and the default destination for the output stream is also the terminal. You saw how to use the `INPUT FROM` and `OUTPUT TO` statements to redirect these input and output streams.

You might find that having just one input stream and one output stream is not enough for particular procedures. That is, you might want to get input from more than one source at the same time or send output to more than one destination at the same time.

Suppose you want to produce a report of the items you have in inventory and you want to send the report to a file. You already know how to use the `OUTPUT TO` statement to redirect the output stream to a file. Suppose that you also want to produce an "exceptions" report at the same time. Any item where the allocated amount is greater than the on-hand amount is an exception. The following figure illustrates this scenario.

Figure 29: Multiple output streams scenario



For items that are exceptions, the procedure needs to send output to a second location. That means you need two different output streams.

You use the DEFINE STREAM statement to define additional streams for a procedure to get input from more than one source simultaneously and send output to more than one destination simultaneously. Streams you name can be operating system files, printers, the terminal, or other non-terminal devices.

Note:

For information regarding the maximum number of streams allowed in OpenEdge applications, see the Input/Output Limits table in *Manage ABL Applications*.

For information about the size limit of operating system files, see the input/output limits section of *Manage ABL Applications*.

The procedure i-dfstr.p uses the two report scenarios shown in the above figure.

i-dfstr.p

```
       DEFINE VARIABLE exception AS LOGICAL   NO-UNDO.
       DEFINE VARIABLE excount   AS INTEGER   NO-UNDO
         LABEL "Total Number of exceptions".
       DEFINE VARIABLE fne       AS CHARACTER NO-UNDO FORMAT "x(12)".
       DEFINE VARIABLE fnr       AS CHARACTER NO-UNDO FORMAT "x(12)".

       DEFINE STREAM exceptions.
       DEFINE STREAM rpt.

/*1*/  SET fnr LABEL "Enter filename for report output" SKIP(1)
           fne LABEL "Enter filename for exception output"
           WITH SIDE-LABELS FRAME fnames.
/*2*/  OUTPUT STREAM rpt TO VALUE(fnr) PAGED.
       OUTPUT STREAM exceptions TO VALUE(fne) PAGED.

/*3*/  DISPLAY STREAM rpt "Item Inventory Report" SKIP(2)
         WITH CENTERED NO-BOX FRAME rpt-frame STREAM-IO.
/*4*/  DISPLAY STREAM exceptions "Item Exception Report" SKIP(2)
         WITH CENTERED NO-BOX FRAME except-frame STREAM-IO.

/*5*/  FOR EACH Item NO-LOCK:
/*6*/    IF Item.OnHand < Item.Allocated THEN DO:
           DISPLAY STREAM exceptions
             Item.ItemNum Item.ItemName Item.OnHand Item.Allocated
             WITH FRAME exitem DOWN STREAM-IO.
           ASSIGN
             excount   = excount + 1
             exception = TRUE.
         END.
/*7*/    DISPLAY STREAM rpt Item.ItemNum Item.ItemName
           WITH NO-LABELS NO-BOX STREAM-IO.
/*8*/    IF exception THEN
           DISPLAY STREAM rpt "See Exception Report".
         exception = FALSE.
       END. /* FOR EACH Item */

/*9*/  DISPLAY STREAM exceptions SKIP(1) excount
         WITH FRAME exc SIDE-LABELS STREAM-IO.
/*10*/ DISPLAY STREAM rpt WITH FRAME exc STREAM-IO.

/*11*/ OUTPUT STREAM rpt CLOSE.
       OUTPUT STREAM exceptions CLOSE.
```

The numbers on the left of the procedure correspond to the following step-by-step descriptions:

1. The SET statement prompts you for the filenames you want to use for the Item Inventory Report and for the Item Exception Report. It stores your answers in the fnr and fne variables, respectively.
2. The OUTPUT STREAM statements open two output streams, named rpt and exceptions. These streams were defined at the start of the procedure with the DEFINE STREAM statement.

   The rpt and exceptions streams are directed to the files whose names you supplied: VALUE(fnr) and VALUE(fne). This means that output can now be sent to either or both of those files.
3. The DISPLAY statement displays the text **Item Inventory Report**. But instead of displaying that text on the terminal, it displays it to the rpt stream. The file you named for the Item Inventory Report contains the text **Item Inventory Report**.
4. This DISPLAY statement also displays text but it uses the exceptions stream. The file you named for the Item Exception Report contains the text **Item Exception Report**.
5. The FOR EACH block reads a single item record on each iteration of the block.
6. If the allocated amount of an item is larger than the on-hand amount of that item then:
   - The DISPLAY statement displays item data to the exceptions stream. After this DISPLAY statement finishes, the file you named for the Item Exception Report contains item data for a single item.
   - The excount counter variable, defined at the start of the procedure, is incremented by 1. The value of this variable is displayed at the end of the procedure so that you know the total number of exception items in inventory.
   - The exception logical variable, defined at the start of the procedure, is set to TRUE.
7. The DISPLAY statement displays some item data to the rpt stream. After this statement finishes, the file you named for the Item Inventory Report contains item data for a single item.
8. If the item is an exception, determined by the value in the exception logical variable, the DISPLAY statement displays the string "See Exception Report" to the rpt stream. That way you know, when looking at the Item Inventory Report, which items are exceptions.
9. The DISPLAY statement displays the value of the excount variable to the exceptions stream. The value of this variable is the total number of exception items.
10. This DISPLAY statement displays the value of the excount variable to the rpt stream. Although the DISPLAY statement does not explicitly say what is being displayed, it does name the same frame, exc, as is used to display excount in the previous DISPLAY statement. That means that the exc frame already contains the excount value. Thus, all this second DISPLAY statement has to do is name the same frame.
11. The OUTPUT STREAM CLOSE statements close the rpt and exception streams, redirecting all further output to the default output destination.

Sharing streams among procedures

If you want several procedures to share the same input or output streams, you can define a shared stream. Another alternative is to use stream object handles.

Defining a shared stream

Use the SHARED parameter of the DEFINE STREAM statement to create a shared stream.

Note: You cannot define or access shared streams in a persistent procedure or in a class file. You can access streams defined in persistent procedures or class files by using stream object handles. See Using stream object handles on page 312 for more information.

For example, the procedures i-sstrm.p and i-dispho.p share the same output stream, phonelist. Notice that phonelist is defined as a shared stream in both procedures.

i-sstrm.p

```
DEFINE NEW SHARED BUFFER xrep FOR salesrep.
DEFINE NEW SHARED STREAM phonelist.

OUTPUT STREAM phonelist TO phonefile.

PAUSE 2 BEFORE-HIDE.

FOR EACH xrep:
  DISPLAY xrep WITH FRAME repname
    TITLE "Creating report for " 2 COLUMNS CENTERED ROW 10.
  DISPLAY STREAM phonelist xrep WITH 2 COLUMNS STREAM-IO.
  RUN i-dispho.p.
END.
```

The i-sstrm.p procedure defines a NEW SHARED STREAM called phonelist. The procedure sends the output from the phonelist stream to a file called phonefile. The procedure also calls the i-dispho.p procedure.

i-dispho.p

```
DEFINE SHARED BUFFER xrep FOR salesrep.
DEFINE SHARED STREAM phonelist.

FOR EACH Customer OF xrep NO-LOCK BY Customer.State:
  DISPLAY STREAM phonelist Customer.CustNum Customer.Name Customer.City
    Customer.State Customer.Phone
    WITH NO-LABELS STREAM-IO.
END.
```

The i-dispho.p procedure defines the SHARED STREAM phonelist, and displays the information from that stream on the screen. (It is more efficient to place the FOR EACH and DISPLAY statements in the i-sstrm.p procedure. They are in a separate procedure here to illustrate shared streams.)

Sharing streams is much like sharing variables because:

- You use a regular DEFINE STREAM statement to define a stream that is available only to the current procedure.
- To define a shared stream, you define the stream as NEW SHARED in the procedure that creates the stream, and as SHARED in all other procedures that use that stream. If you do not explicitly close the stream, ABL closes it automatically at the end of the procedure in which you defined it.
- You define the stream as NEW GLOBAL when you want that stream to remain available even after the procedure that contains the DEFINE NEW GLOBAL SHARED STREAM statement ends.

For more information, see the DEFINE STREAM statement in *ABL Reference*.

Using stream object handles

Stream object handles allow you access named streams defined in routines (procedures, persistent procedures, user-defined functions, and methods of classes). The ABL Virtual Machine (AVM) implicitly creates a stream object when an application first tries to get the handle to a named stream, and deletes it when the routine that defines the stream terminates.

Note: You **cannot** use handles for the unnamed streams. By default, ABL provides two unnamed streams for input and output.

Like other ABL handle-based objects, you access stream attributes and methods using the object handle. Unlike other ABL handle-based objects, you use a DEFINE(and not a CREATE) statement in conjunction with streams.

All ABL statements that take a STREAM parameter can take STREAM-HANDLE parameter. For example:

```
DOWN [STREAM stream| STREAM-HANDLE handle][expression]
{[frame-phrase]}
```

The STREAM-HANDLE option takes an expression that evaluates to a stream handle. An application can use the STREAM*name* option or the STREAM-HANDLE*handle* option, but cannot use both in the same statement.

The following example shows how you define a stream and access the handle to the stream:

```
DEFINE VARIABLE hStream AS HANDLE NO-UNDO.
DEFINE STREAM myStream.

hStream = STREAM mySTream:HANDLE.
```

For more information, see the DEFINE STREAM statement and the stream object handle section in *ABL Reference*.

Summary of opening and closing streams

The following table describes how you establish, open, use, and close default streams and streams you name.

Table 41: Using streams

Action	Unnamed streams	Named streams
Establish the stream	By default, each procedure gets one unnamed input stream and one unnamed output stream.	You define the stream explicitly by using one of these statements: DEFINE STREAM, DEFINE NEW SHARED STREAM, DEFINE SHARED STREAM, DEFINE NEW GLOBAL SHARED STREAM. For more information, see the "DEFINE STREAM statement".

Action	Unnamed streams	Named streams
Open the stream	Automatically opened, using the output destination to which the calling procedure's unnamed stream is directed and the input source from which the calling procedure's input is read. You can also explicitly name a destination or source by the using "OUTPUT TO statement" or the "INPUT FROM statement".	You open the stream explicitly by using: • `OUTPUT { STREAM stream | STREAM-HANDLE handle } TO` • `INPUT { STREAM stream | STREAM-HANDLE handle } FROM`
Use the stream	All data-handling statements use the stream by default.	Use the name or the handle of the opened stream in the data-handling statement that will use the stream.
Close the stream	Automatically closed at the end of the procedure that opened it. You can also explicitly close it with the "OUTPUT CLOSE statement" or the "INPUT CLOSE statement".	Local streams are automatically closed at the end of the procedure. Shared streams are automatically closed when the procedure that defined the stream as `NEW` ends. Global streams are closed at the end of the OpenEdge session. You can also explicitly close named streams by using the `INPUT CLOSE` or `OUTPUT CLOSE` statement or by opening the stream to a new destination or from a new source.

Initially, a default input stream has as its source the most recent source specified in the calling procedure or, if there is no calling procedure, the terminal. The default output stream has as its destination the most recent destination specified in the calling procedure or, if there is no calling procedure, the terminal. If you are running a procedure in batch or background, you must explicitly indicate a source and/or destination.

When an unnamed stream is closed (either automatically or explicitly), it is automatically redirected to its previous destination (the destination of the procedure it is in). If the stream is not in a procedure, the stream is redirected to or from the terminal.

When you close a named stream, you can no longer use that stream until it is reopened. When you close an input stream associated with a file and then reopen that stream to the same file, input starts from the beginning of the file.

Processes as input and output streams (NT and UNIX only)

You can import data into ABL from a process or pipe data from ABL to another process using one of the following statements:

- `INPUT THROUGH` statement to import data into ABL from another process.
- `OUTPUT THROUGH` statement to pipe data from ABL to another process.
- `INPUT–OUTPUT THROUGH` statement to pipe the output of a process into an OpenEdge procedure and to pipe data from OpenEdge back to that same process.

  This allows two-way communication to a program written in C or any other language. You might use this capability to do specialized calculations on data stored in an OpenEdge database or entered during an OpenEdge session.

For more information on these statements, see *ABL Reference*.

Portability issues

To ensure portability, avoid using the `INPUT THROUGH` and `OUTPUT THROUGH` statements. In place of `OUTPUT THROUGH`, use the `OUTPUT TO` statement with an escape to the operating system, as shown in the following example:

```
OUTPUT TO file1.
. . .
OUTPUT CLOSE.
CASE OPSYS:
  WHEN "UNIX" OR "Win32" THEN
    OS-COMMAND program-name < file1.
  OTHERWISE
    MESSAGE OPSYS "Operating system not supported".
END CASE.
```

In this example, you send the data to `file1`. Then you use the appropriate operating system statement to escape to the operating system. Once at the operating system level, you run the program against the data in `file1`.

In place of `INPUT THROUGH`, use an escape to the operating system with the `INPUT FROM` statement, as shown in the following example:

```
CASE OPSYS:
  WHEN "UNIX" THEN OS-COMMAND program-name > file1.
INPUT FROM file1.
. . .
INPUT CLOSE.
```

First, you use the operating system statement to escape to the operating system. Once at the operating system level, you run a program to create the data in `file1`. Then you use the `INPUT FROM` statement to retrieve the data in `file1`.

For information about the size limit of data files, see the input/output limits section of *Manage ABL Applications*.

I/O redirection for batch jobs

You can use the < and > symbols on the OpenEdge command line to redirect I/O for batch jobs. For details, see the entry for the Batch (–b) startup parameter in *Startup Command and Parameter Reference*.

Reading the contents of a directory

Sometimes, rather than reading the contents of a file, you want to read a list of the files in a directory. You can use the `OS-DIR` option of the `INPUT FROM` statement for this purpose.

Each line read from `OS-DIR` contains three values:

- The simple (base) name of the file.
- The full pathname of the file.

- A string value containing one or more attribute characters. These characters indicate the type of the file and its status. Every file has one of the following attribute characters:
  - **F** — Regular file or FIFO pipe
  - **D** — Directory
  - **S** — Special device
  - **X** — Unknown file type

  In addition, the attribute string for each file might contain one or more of the following attribute characters:
  - **H** — Hidden file
  - **L** — Symbolic link
  - **P** — Pipe file

The tokens are returned in the standard ABL format that can be read by the `IMPORT` or `SET` statements.

The following example uses the `OS-GETENV` function to find the path of the `DLC` directory. It then uses the `OS-DIR` option of `INPUT FROM` to read the contents of the directory:

i-osdir.p

```
DEFINE VARIABLE attr-list  AS CHARACTER NO-UNDO
  FORMAT "x(4)" LABEL "Attributes".
DEFINE VARIABLE file-name  AS CHARACTER NO-UNDO
  FORMAT "x(16)" LABEL "File".
DEFINE VARIABLE search-dir AS CHARACTER NO-UNDO.

search-dir = OS-GETENV("DLC").
INPUT FROM OS-DIR(search-dir).

REPEAT:
  SET file-name ^ attr-list
    WITH WIDTH-CHARS 70 USE-TEXT TITLE "Contents of " + search-dir.
END.

INPUT CLOSE.
```

In `i-osdir.p`, only the base name of the file and attribute string are read from `OS-DIR`. The caret (^) is used in the `SET` statement to skip over the pathname of the file.

For more information on the `OS-DIR` option, see the `INPUT FROM` Statement reference entry in *ABL Reference*.

You can find additional information on a single file by using the `FILE-INFO` system handle. To use the `FILE-INFO` handle, first assign the pathname of an operating system file to the `FILE-INFO:FILE-NAME` attribute. You can then read other `FILE-INFO` attributes. For example, the `i-osfile.p` procedure prompts for the pathname of a file and then uses the `FILE-INFO` handle to get information on that file.

i-osfile.p

```
DEFINE VARIABLE os-file AS CHARACTER NO-UNDO FORMAT "x(60)" LABEL "File".

REPEAT:
  SET os-file WITH FRAME osfile-info.
  FILE-INFO:FILE-NAME = os-file.

  DISPLAY FILE-INFO:FULL-PATHNAME FORMAT "x(60)" LABEL "Full Path"
    FILE-INFO:PATHNAME FORMAT "x(60)" LABEL "Path"
    FILE-INFO:FILE-TYPE LABEL "Type"
    WITH FRAME osfile-info SIDE-LABELS TITLE "OS File Info".
END.
```

For more information, see the FILE-INFO System Handle reference entry in *ABL Reference*.

Performing code-page conversions

Computer systems store text data using character codes, typically 7 or 8–bit numeric codes that map to specific visual character representations. The series of character codes that make up the character set that a system uses is referred to as a code page.

ABL provides a character set management facility to automatically convert data between the code pages of different data sources and destinations (targets). In general, the supported data sources and targets for code page conversion include memory, streams, and databases. You can specify default code page conversions for a session using conversion tables and startup parameters to specify the code page for each data source and target. For more information on this character set facility, see *Internationalize ABL Applications*.

ABL also allows you to perform I/O that explicitly converts data from one code page to another in order to facilitate I/O between data sources and destinations intended for different computer systems or components. You can specify the name of the code page for a data source and target as parameters to several statements and functions.

To convert between source and target characters or strings in memory, you can specify code page parameters in these functions:

- ASC
- CHR
- CODEPAGE-CONVERT

To convert data input and output, you can specify code page parameters in these statements:

- INPUT FROM (input source to memory target)
- OUTPUT TO (memory source to output target)

To convert piped input and output, you can specify code page parameters in these statements:

- INPUT THROUGH (program source to memory target)
- OUTPUT THROUGH (memory source to program target)
- INPUT-OUTPUT THROUGH (program source to program target)

These statements and functions take available code page name parameters as character expressions (for example, "ibm850"). The code page names you specify must be defined in your ABL conversion map file (`convmap.cp`, by default). Also, the source and target conversion tables must be defined in the conversion map file for each code page to support the specified conversions. For more information on building a conversion map file, see *Internationalize ABL Applications*.

For example, the `i-codpag.p` procedure writes to two output streams using two different code pages.

i-codpag.p

```
DEFINE VARIABLE xname NO-UNDO LIKE Customer.Name INITIAL ?.

DO WHILE xname <> "":
  ASSIGN xname = "".
  DISPLAY xname LABEL "Starting Name to German List"
    WITH FRAME a SIDE-LABELS.
  SET xname WITH FRAME a.
  IF xname <> "" THEN DO:
    OUTPUT TO german.txt
      CONVERT SOURCE "iso8859-1" TARGET "german-7-bit".

    FOR EACH Customer NO-LOCK WHERE Customer.Name >= xname
      BY Customer.Name:
      DISPLAY Customer WITH STREAM-IO.
    END.
    OUTPUT CLOSE.

    OUTPUT TO swedish.txt
      CONVERT SOURCE "iso8859-1" TARGET "swedish-7-bit".

    FOR EACH Customer WHERE Customer.Name < xname BY Customer.Name:
      DISPLAY Customer WITH STREAM-IO.
    END.
    OUTPUT CLOSE.
  END.
END.
```

For this example, assume that the internal code page is "iso8859–1". If a customer name from the `sports2000` database is greater than or equal to a prompted name (xname), then the procedure writes the corresponding `Customer` record to the file, `german.txt`, using the "german–7–bit" code page. Otherwise, it writes the corresponding `Customer` record to the file, `swedish.txt`, using the "swedish–7–bit" code page. These conversions are handled by the conversion tables provided with ABL.

For more information on specifying code page parameters, see the reference entry for each statement or function that supports code page conversion in *ABL Reference*.

Converting nonstandard input files

The input files used in previous examples contained data that was in a very specific format. For example, see Input from files on page 306.

When using a data file as an input source, the AVM, by default, expects that file to conform to the following standards:

- One or more spaces must separate each field value.
- Character fields that contain embedded blanks must be surrounded by quotes (" ").
- Any quotes in the data must be represented by two quotes (" ").

What if you need to deal with an input file that does not conform to these standards? For example, you might have a file in which each field is on a separate line; or where fields are separated by commas instead of spaces; or where the fields have no special delimiters, but appear at specific column locations.

ABL provides two strategies for dealing with such files:

- Use the QUOTER utility to convert the file to the standard ABL format. You can then use the normal ABL frame-based input statements, such as SET, to read data from the file.
- Use the IMPORT statement to read from the file.

Which method you choose depends on your particular circumstances. For example:

- The frame-based statements, such as SET, validate the format of the incoming data; IMPORT does not. In some cases you might want this validation and in others you might not.
- If you expect to read the same file repeatedly, preprocessing that file once with QUOTER might be worthwhile. However, if your code reads from different files, or the content of the file is subject to change, you might want to avoid repeated preprocessing by using IMPORT instead.

The following section explains how to use the IMPORT statement. For information about the QUOTER utility, see Command and Utility Reference on page 701.

Importing and exporting data

Sometimes you send data to a file knowing that it will be used later by an ABL procedure. If so, then you also know that the data file must be in standard format with character fields surrounded by quotes. Therefore, instead of just redirecting the output to the file and using the DISPLAY statement to send output to that file, you might use the EXPORT statement.

Note: Do not confuse the EXPORT statement with the EXPORT method of the SESSION handle. The EXPORT statement converts data from one format to another while redirecting it. The EXPORT method of the SESSION handle adds procedure names to the export list (a list of procedures a requesting procedure can access) of a OpenEdge application server.

Using the EXPORT statement

The EXPORT statement sends data to a specified output destination, formatting it in a way that can be easily used by another ABL procedure. For example, i-export.p writes customer information to a file.

i-export.p

```
OUTPUT TO i-datfl6.d.
FOR EACH Customer NO-LOCK:
  EXPORT Customer.CustNum Customer.Name Customer.SalesRep.
END.
OUTPUT CLOSE.
```

The output from i-export.p is written to i-datfl6.d.

i-datf16.d

```
1 "Lift Line Skiing" "HXM"
2 "Urpon Frisbee" "DKP"
3 "Hoops Croquet Comp" "HXM"
4 "Go Fishing Ltd" "SLS"
5 "Match Point Tennis" "JAL"
       .
       .
       .
```

Now this file is ready to be used as an input source by another ABL procedure. There is no need to process it through QUOTER.

By default, the EXPORT statement uses the space character as a delimiter between fields. You can use the DELIMITER option of the EXPORT statement to specify a different delimiter.

For example, i-exprt2.p writes to a file in which field values are separated by commas.

i-exprt2.p

```
OUTPUT TO i-datfl7.d.
FOR EACH Customer NO-LOCK:
  EXPORT DELIMITER "," Customer.CustNum Customer.Name Customer.SalesRep.
END.
OUTPUT CLOSE.
```

The output from i-exprt2.p is written to i-datfl7.d.

i-datf17.d

```
1,"Lift Line Skiing","HXM"
2,"Urpon Frisbee","DKP"
3,"Hoops Croquet Comp","HXM"
4,"Go Fishing Ltd","SLS"
5,"Match Point Tennis","JAL"
       .
       .
       .
```

You can read this file by using the DELIMITER option of the IMPORT statement. More likely, you would prepare a file like this to be read by another application.

For more information on the EXPORT statement, see *ABL Reference*.

Using the PUT statement

If you need to prepare a data file in a fixed format, perhaps for use by another system, you can use the PUT statement, as shown in the i-putdat.p procedure.

i-putdat.p

```
OUTPUT TO i-datfl8.d.
FOR EACH Customer NO-LOCK:
  PUT Customer.CustNum AT 1 Customer.Name AT 10 Customer.SalesRep AT 40 SKIP.
END.
OUTPUT CLOSE.
```

The output from i-putdat.p is written to i-datfl8.d.

i-datf18.d

```
     1        Lift Line Skiing         HXM
     2        Urpon Frisbee            DKP
     3        Hoops Croquet Comp       HXM
     4        Go Fishing Ltd           SLS
     5        Match Point Tennis       JAL
            .
            .
            .
```

The PUT statement formats the data into the columns specified with the AT options. Only the data is output: there are no labels and no box. The SKIP option indicates that you want each customer's data to begin on a new line.

For more information on the PUT statement, see *ABL Reference*.

Using the IMPORT statement

The IMPORT statement is the counterpart of the EXPORT statement. It reads an input file into ABL procedures, one line at a time.

The i-import.p procedure shows IMPORT reading the file exported by the i-export.p procedure.

i-import.p

```
INPUT FROM i-datfl6.d.
REPEAT:
  CREATE Customer.
  IMPORT Customer.CustNum Customer.Name Customer.SalesRep.
END.
INPUT CLOSE.
```

This relies on the input being space separated. You can also use the DELIMITER option of the IMPORT statement to read a file with a different separator.

For example, i-imprt2.p reads the file produced by i-exprt2.p shown in Using the EXPORT statement on page 319.

i-imprt2.p

```
INPUT FROM i-datfl7.d.
REPEAT:
  CREATE customer.
  IMPORT DELIMITER "," Customer.CustNum Customer.Name Customer.SalesRep.
END.
INPUT CLOSE.
```

This example reads one line at a time from i-datfl7.d into the character-string variable data. It then breaks the line into discrete values and assigns them to the fields of a Customer record.

Although the IMPORT statement is used primarily to read data in the standard format written by the EXPORT statement. However, you can use the UNFORMATTED and DELIMITER options of IMPORT to read data in non-standard formats.

When you use the UNFORMATTED option, the IMPORT statement reads one line from the input file. For example, suppose your input file is formatted as shown in i-datf12.d.

i-datf12.d

```
90
Wind Chill Hockey
BBB
91
Low Key Checkers
DKP
92
Bing's Ping Pong
SLS
```

The lines containing CustNum and SalesRep values can be read with normal IMPORT statements. However, if you try to read the Customer Name values with a normal IMPORT statement, only the first word of each Name is read—the space character is treated as a delimiter. To prevent this, read the Name with the UNFORMATTED option, as in i-impun1.p.

i-impun1.p

```
INPUT FROM i-datfl2.d.
REPEAT:
  CREATE Customer.
  IMPORT Customer.CustNum.
  IMPORT UNFORMATTED Customer.Name.
  IMPORT Customer.SalesRep.
END.
INPUT CLOSE.
```

Now, suppose each line of the file contained a CustNum, Name, and SalesRep value, but no special delimiters are used. Instead, the fields are defined by their position within the line as shown in idatf13.d.

i-datf13.d

```
90 Wind Chill Hockey  BBB
91 Low Key Checkers   DKP
92 Bing's Ping Pong   SLS
```

In i-datfl3.d, the first three character positions in each line are reserved for the CustNum value, the next 17 positions for the Name value, and the last three for the SalesRep value. Space characters may occur between fields, but they may also occur within a field value. To process this file with the IMPORT statement, use the UNFORMATTED option to read one line at a time, as shown in i-impun2.p.

i-impun2.p

```
DEFINE VARIABLE file-line AS CHARACTER NO-UNDO.

INPUT FROM i-datfl3.d.
REPEAT:
  CREATE Customer.
  IMPORT UNFORMATTED file-line.
  ASSIGN
    Customer.CustNum  = INTEGER(SUBSTRING(file-line, 1, 2))
    Customer.Name     = TRIM(SUBSTRING(file-line, 4, 17))
    Customer.SalesRep = SUBSTRING(file-line, 22, 3).
END.
INPUT CLOSE.
```

After `i-impun2.p` reads each line, it uses the `SUBSTRING` function to break the line into field values. It then assigns these values to the appropriate fields in the customer record.

Note: If a line in your input file ends with a tilde (~), ABL interprets that as a continuation character. This means, that line and the following line are treated as a single line. Therefore, the `IMPORT` statement with the `UNFORMATTED` option reads both lines into a single variable.

What if fields values are separated by a delimiter other than the space character? For example, in `i-datfl4.d`, field values are separated by commas.

i-datf14.d

```
90,Wind Chill Hockey,BBB
91,Low Key Checkers,DKP
92,Bing's Ping Pong,SLS
```

You could use the `UNFORMATTED` option of the `IMPORT` statement to read this file one line at a time and then use the `INDEX` function to locate the commas and break the line into field values. Another solution is to use the `DELIMITER` option of the `IMPORT` statement as shown in `i-impun3.p`.

i-impun3.p

```
INPUT FROM i-datfl4.d.
REPEAT:
  CREATE Customer.
  IMPORT DELIMITER "," Customer.Custum Customer.Name Customer.SalesRep.
END.
INPUT CLOSE.
```

In `i-impun3.p`, the `DELIMITER` option specifies that field values are separated by commas rather than by spaces. Therefore, the `IMPORT` statement parses each line correctly and assigns each value to the appropriate field.

Note: You can only specify a single character as a delimiter. If the value you give with the `DELIMITER` option is longer than one character, then only the first character is used.

For more information on the `IMPORT` statement, see *ABL Reference*.

8

Colors and Fonts

ABL (Advanced Business Language) allows you to control the colors and fonts that are displayed in a widget. The extent of this control depends on your user interface and how you manage colors and fonts in your application.

For details, see the following topics:

- Making colors and fonts available to an application
- Assigning colors and fonts to a widget
- Assigning colors and fonts to ActiveX Automation objects and ActiveX controls
- Color and font inheritance
- Color in character interfaces
- Colors in windows in graphical interfaces
- Managing colors and fonts in graphical applications
- Allowing the user to change colors and fonts
- Accessing the current color and font tables
- Retrieving and changing color and font definitions
- Managing application environments
- Managing color display limitations

Making colors and fonts available to an application

OpenEdge applications typically access a subset of the system's colors and fonts. This subset is specified in the following places:

- On Windows, in the registry or in an initialization file
- On UNIX, in the PROTERMCAP file

These environment settings establish a one-to-one mapping between a range of integers and the system's colors and fonts. These integers, in turn, correspond to entries in internal color and font tables. For example, on Windows, OpenEdge, as installed, maps the color red to the integer 4. To assign red to a widget, an application assigns the value 4 to the appropriate widget attribute. Thus, the widget is assigned the fourth color from the internal color table.

For graphical interfaces, you can specify up to 256 colors and 256 fonts. However, it is also possible to define an arbitrary color, expanding beyond the 256 colors defined in the color table, using the RGB-VALUE function. The font for an ActiveX control is set through standard font properties and has no programmatic relationship to the font table. For information about using colors and fonts for ActiveX controls, see the topics that discusses Active X controls in ActiveX Control Support on page 551.

For character interfaces, you can specify up to 128 colors. When OpenEdge starts up, it loads the color and font definitions from the specified environment into internal color and font tables. The numbers for both your color and font definitions must increase by 1, starting from 0. OpenEdge stops loading its color or font table at the first skipped value.

For example, if colors 0, 1, 2, and 4 are defined, OpenEdge loads only colors 0 through 2 because color 3 is missing. If colors 1, 2, and 3 are defined, OpenEdge loads no colors because the definitions start after 0. The order of color and font definition does not matter; it is only important that a complete sequence of color and font numbers is defined.

For more information on specifying and editing environments for graphical and character interfaces, see *Manage ABL Applications*.

OpenEdge default colors

The environment that comes with your OpenEdge installation specifies 16 colors. The following table shows these colors and the integers you use to reference them in an application. Note that for backward compatibility, the 16 colors are the same as those available in Version 6 of Progress.

Caution: The OpenEdge Application Development Environment (ADE) reserves these 16 colors (0 through 15) defined in your environment. If you change the mappings of these colors, the ADE tools might not function properly. You can add your own application colors beginning with number 16.

Table 42: OpenEdge default colors

Color number	Windows RGB values	Actual color defined
0	0, 0, 0	Black
1	0, 0, 128	Dark blue
2	0, 128, 0	Dark green
3	0, 128, 128	Blue green

Color number	Windows RGB values	Actual color defined
4	128, 0, 0	Red
5	128, 0, 128	Purple
6	128, 128, 0	Olive
7	128, 128, 128	Gray
8	192, 192, 192	Light gray
9	0, 0, 255	Blue
10	0, 255, 0	Green
11	0, 255, 255	Turquoise
12	255, 0, 0	Red
13	255, 0, 255	Pink
14	255, 255, 0	Yellow
15	255, 255, 255	White

OpenEdge default fonts

The default OpenEdge environment specifies a set of fonts to be used with the OpenEdge Tools. On Windows, these fonts are MS Sans Serif (a proportional font) and Courier New (a fixed-space font). If you have not added custom fonts to your environment, OpenEdge uses these default fonts for your application. OpenEdge uses the default proportional font for alphanumeric data, and the default fixed font for integer fields and for fields with format strings containing fill characters (such as 9 or X).

Caution: The OpenEdge ADE reserves the 8 fonts (0 through 7) defined in your environment. If you change the mappings of these fonts, the ADE tools might not function properly. You can add your own application fonts, beginning with number 8.

Assigning colors and fonts to a widget

You can assign colors and fonts to a widget either in the widget definition statement or at runtime after the widget is displayed. Use the FGCOLOR, BGCOLOR, DCOLOR, PFCOLOR, and FONT options at definition time and the FGCOLOR, BGCOLOR, DCOLOR, PFCOLOR, and FONT attributes at runtime.

ABL uses the foreground color you specify for any values that appear in the widget; ABL uses the background color for the area around the widget values.

Note: For rectangle widgets, ABL uses the foreground color for the edge and the background color to fill the interior.

ABL uses the font you specify for any text that appears within the widget.

Note that FGCOLOR, BGCOLOR, and FONT apply to graphical interfaces only, and DCOLOR and PFCOLOR apply to character interfaces only. For more information on specifying color in character interfaces, see Color in character interfaces on page 330.

The i-clrfnt.p procedure demonstrates how to initialize colors and fonts at widget definition and how to change them dynamically at runtime.

i-clrfnt.p

```
DEFINE VARIABLE fgc_frm  AS INTEGER NO-UNDO BGCOLOR 15 FGCOLOR 0
  VIEW-AS SLIDER MAX-VALUE 15 MIN-VALUE 0.
DEFINE VARIABLE bgc_frm  AS INTEGER NO-UNDO BGCOLOR 15 FGCOLOR 0
  VIEW-AS SLIDER MAX-VALUE 15 MIN-VALUE 0.
DEFINE VARIABLE font_frm AS INTEGER NO-UNDO FONT 1
  VIEW-AS SLIDER MAX-VALUE 15 MIN-VALUE 0.
DEFINE BUTTON quitbtn LABEL "QUIT" BGCOLOR 4 FGCOLOR 15.

FORM
   SKIP (1) "Form Foreground" AT 7 SKIP
   fgc_frm AT 6 SKIP (1)
   "Form Background" AT 7 SKIP
   bgc_frm AT 6 SKIP (1)
   "Form Font" AT 10 SKIP
   font_frm AT 6 SKIP(2)
   quitbtn AT 12 SKIP(1)
   WITH FRAME x TITLE "Color and Font Test" NO-LABELS CENTERED ROW 2
     WIDTH-CHARS 50.

ASSIGN
  FRAME x:RULE-Y     = 282
  fgc_frm:MAX-VALUE  = COLOR-TABLE:NUM-ENTRIES - 1
  bgc_frm:MAX-VALUE  = COLOR-TABLE:NUM-ENTRIES - 1
  font_frm:MAX-VALUE = FONT-TABLE:NUM-ENTRIES - 1.

ON VALUE-CHANGED OF fgc_frm
  FRAME x:FGCOLOR = INPUT FRAME x fgc_frm.

ON VALUE-CHANGED OF bgc_frm
  FRAME x:BGCOLOR = INPUT FRAME x bgc_frm.

ON VALUE-CHANGED OF font_frm
  FRAME x:FONT = INPUT FRAME x font_frm.

ENABLE ALL WITH FRAME x.

WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW OR CHOOSE OF quitbtn.
```

This procedure creates three sliders representing foreground color, background color, and font. The maximum value of each slider is determined by examining the NUM-ENTRIES attributes of the COLOR-TABLE and FONT-TABLE system handles. As you move the appropriate trackbar, the foreground color, background color, or font of the interface changes. Note that, because the procedure does not explicitly set the size of any of the sliders, the ABL Virtual Machine (AVM) resizes them at runtime as you change the font.

You can assign colors and fonts to all widgets, with the following exceptions:

- In Windows, to assign colors to buttons, menus, and window titles, you must use the **Display Properties** dialog box in the **Control Panel**. You cannot change these colors in your OpenEdge application.
- In all environments, images and rectangles have no text and, therefore, cannot take a font assignment.

For more information on the COLOR-TABLE and FONT-TABLE handles, and on the FGCOLOR, BGCOLOR, DCOLOR, PFCOLOR, and FONT options and attributes, see *ABL Reference*. For more information on using the COLOR-TABLE and FONT-TABLE handles for run-time color and font management, see Accessing the current color and font tables on page 337.

Assigning colors and fonts to ActiveX Automation objects and ActiveX controls

Properties associated with ActiveX Automation objects and ActiveX controls allow you to define unique characteristics, such as color and font, for each object. For example, to set colors for an ActiveX control, you need to set one or more of the color properties of the control to some RGB value. There are three ways to obtain an RGB value. You can use the RGB–VALUE function, use the COLOR-TABLE:GET-RGB-VALUE( ) method, or you can get the value from some color property of an ActiveX control or an ActiveXAutomation object. For fonts, an ActiveX Automation Server or ActiveX control generally provide a Font object that allows you to specify font properties.

For further information about assigning colors and fonts to ActiveX controls and ActiveX Automation objects, see ActiveX Automation Support on page 543 and ActiveX Control Support on page 551.

Color and font inheritance

If you do not specify colors or a font for a widget, ABL assigns colors and a font using the following rules of precedence:

1. Field-level widgets inherit the colors and font of the containing frame.
2. Frame-level widgets inherit the default colors and fonts specified in the environment.

   Note: In the environment, you can specify two default fonts but you cannot specify default widget colors.

3. Otherwise, frame-level widgets inherit the default colors and fonts specified for the operating system.

   Note: On Windows, the default foreground color is color Window Text. For two-dimensional widgets, the default background color is color Window. For three-dimensional widgets, the default background color is color Button Face. These colors are configurable using the Control Panel.

Note that frames do **not** inherit colors and fonts from the containing window. If you do not specify the font for a frame, ABL uses the default font, not the font of the window. This is because ABL determines the frame layout at compile time when the window's colors and fonts (determined at runtime) are not yet available.

Overriding default color inheritance

If you specify colors for a frame-level widget, but not its field-level widgets, the field-level widgets inherit the colors of the containing frame, by default. ABL provides widget attributes and options that allow you to override this default color inheritance for the following field-level widgets: BROWSE, COMBO-BOX (all types), EDITOR, FILL-IN (NATIVE and Enabled), and SELECTION-LIST.

You can use the INHERIT-FGCOLOR and INHERIT-BGCOLOR attributes to control whether the field-level widgets listed above inherit the foreground and background colors of their containing frame. Setting these attributes to TRUE allows these field-level widgets to inherit the foreground and background colors of their containing frame. Setting these attributes to FALSE prevents these field-level widgets from inheriting the foreground and background colors of their containing frame. You can also set these attributes on the SESSION system handle to specify the foreground and background color inheritance for all frame-level widgets created in the session.

When defining static frame-level widgets, you can control foreground and background color inheritance on a frame-by-frame basis by specifying the following options on the Frame phrase:

- Use the INHERIT-FGCOLOR and INHERIT-BGCOLOR options to allow field-level widgets in the frame to inherit the frame's foreground and background colors.
- Use the NO-INHERIT-FGCOLOR and NO-INHERIT-BGCOLOR options to prevent field-level widgets in the frame from inheriting the frame's foreground and background colors.

Color in character interfaces

Color in a character interface differs from color in a graphical interface in these ways:

- In character interfaces, the BGCOLOR and FGCOLOR attributes have the unknown value (?).
- Each location in the color table contains a foreground/background color pair.
- DCOLOR specifies a foreground/background color pair that a widget uses when it displays data.
- PFCOLOR specifies a foreground/background color pair that a widget uses when it prompts for data.

When designing your interface, make sure that you specify different color pairs for DCOLOR and PFCOLOR.

Widget states and color

Frames, dialog boxes, and rectangles use only DCOLOR. For all other widgets, ABL considers the widget's category and its state to determine when to use the display colors (DCOLOR) and when to use the prompt-for colors (PFCOLOR). The categories are text input widgets (fill-ins and editors) and selectable widgets (all other widgets). The widget states are insensitive, sensitive, and focus.

The following table shows how ABL assigns colors based on the state and category of widget.

Table 43: Colors used in character interfaces

Widget state	Text input widget	Selectable widget
Insensitive	DCOLOR	DCOLOR
Sensitive	PFCOLOR	DCOLOR
Focus	PFCOLOR	PFCOLOR

Note that for repaint, look, and usability reasons, text input widgets have the same colors for the sensitive and focus states but different colors for the insensitive state. This is in contrast to selectable widgets, which must change colors when they receive focus.

Color specification

In character interfaces, ABL reserves color table locations 0 through 4 for the following foreground/background color pairs:

- Color 0 holds colors used as the NORMAL color by ABL.
- Color 1 holds the colors used as the INPUT color by ABL. As installed, this is underline mode.
- Color 2 holds the colors used as the MESSAGE color by ABL. As installed, this is reverse video.
- Color 3 holds the colors used for high-intensity mode. This mode is not available for all terminals.

- Color 4 holds the colors used for blink mode. This mode is not available for all terminals.

By default, text-input widgets use the `INPUT` colors when in prompt-for mode. Note, however, that when displaying an editor widget with scroll bars to display read-only text, using `NORMAL` rather than `INPUT` for prompt-for mode might make more visual sense to the user.

In the `PROTERMCAP` file, you can specify additional application color pairs from color 5 to 127. For more information, see *Manage ABL Applications*.

Colors in windows in graphical interfaces

On Windows in graphical interfaces, your system handles color differently depending on whether it supports the Palette Manager.

Systems with the Palette Manager

If your system supports the Palette Manager, you can define a palette with some number of simultaneous colors (usually 256). Of these colors, 20 are normally reserved for standard system colors. The others are custom colors that you can define and modify. The exact numbers of color definitions and reservations by the system depend on your display driver.

Managing colors for multiple windows

Although you can define a separate palette for each window, sharing a single palette among several windows maximizes environment efficiency and eliminates any color contention between the sharing windows. For more information, see Managing application environments on page 340.

Managing colors for bitmap images

If you use bitmap images saved in 256–color mode, Windows tries to create an individual color map for the image each time it is realized in ABL. This has the following effects:

- The background flashes as the color palette is loaded for each 256–color image, unless the color map is identical to the one previously loaded
- Performance degrades

To reduce or eliminate these effects, convert your images from 256 colors to 16 colors or recreate them with 16 colors at a time.

Systems without the Palette Manager

On systems that do not support the Palette Manager, Windows can only display 16 standard colors. However, if you request another color, Windows tries to create that color by *dithering*; that is, by filling a space with dots of different hues. This might work well, for example, in a frame background. However, if a field-level widget, such as a fill-in or radio set, inherits a dithered color, only one hue is used. The result might be undesirable. Therefore, avoid using dithered colors for field-level widgets.

Managing colors and fonts in graphical applications

Graphical interfaces provide a variety of techniques to help manage color and font definitions for an application. Using these techniques, ABL allows a graphical application to manage colors and fonts at four levels of run-time operation:

- **User access to the color and font tables** — Allows the user to change the definitions for current dynamic color and font table entries using the SYSTEM-DIALOG COLOR and SYSTEM-DIALOG FONT statements. For more information, see Allowing the user to change colors and fonts on page 332.
- **Procedure access to the color and font tables** — Examines and changes the definitions for color table entries in the current environment using the COLOR-TABLE handle, and examining height and width characteristics of font table entries in the current environment using the FONT-TABLE handle. Also, this section addresses setting the definition for arbitrary colors using the RGB-VALUE function. For more information, see Accessing the current color and font tables on page 337.
- **Procedure access to color and font definitions** — Reads and writes color and font definitions for the current environment using the GET-KEY-VALUE and PUT-KEY-VALUE statements. For more information, see Retrieving and changing color and font definitions on page 339.
- **Procedure replacement of the current environment** — Replaces the application's current color and font tables with tables from a different environment using the LOAD, USE, and UNLOAD statements. For more information, see Managing application environments on page 340.

Allowing the user to change colors and fonts

In graphical interfaces, ABL allows you to invoke native common dialogs for color and font selection in ABL using the SYSTEM-DIALOG COLOR and the SYSTEM-DIALOG FONT statements. Using these dialogs, you can allow the user to choose a new value for a color or font number. Once the user chooses a value, ABL stores the choice in the color or font table location specified by the number. All visible widgets defined with the specified color or font number immediately redisplay according to the user selection.

Establishing dynamic colors

You can use the SYSTEM-DIALOG FONT statement to change any font in the font table. However, you can use the SYSTEM-DIALOG COLOR statement to update only those colors that are dynamic. To make a color dynamic, use the COLOR-TABLE system handle. For example:

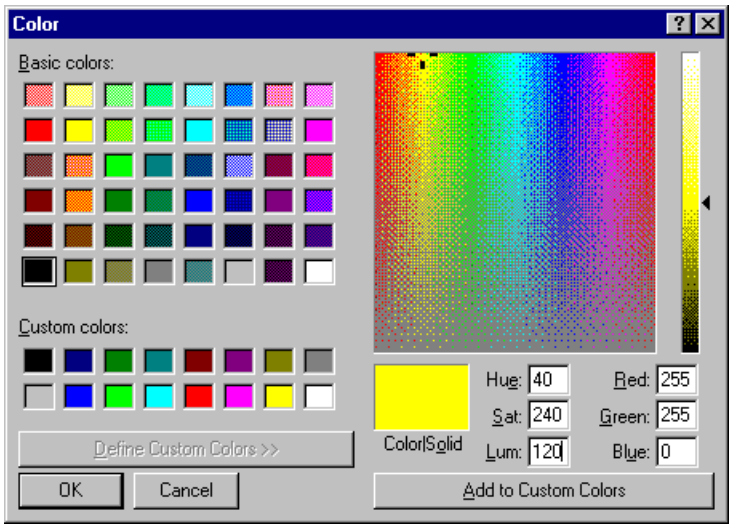
```
DEFINE VARIABLE status-ok AS LOGICAL NO-UNDO.
status-ok = COLOR-TABLE:SET-DYNAMIC(n, TRUE).
```

In this example, *n* is the color table entry you want to make dynamic. For more information on the COLOR-TABLE handle, see *ABL Reference* and Accessing the current color and font tables on page 337.

Color dialog box

The following figure shows the Windows system color dialog box.

Figure 30: Windows system color dialog box



The Windows color dialog provides several ways to specify a new color:

- **Custom colors** — You can create a palette of 16 custom colors. To specify a color, you select the color box from the 16 available custom colors. The specified color appears in the ColorSolid box. To change a custom color, select one of the Basic Colors, or specify a color by using the Numeric Colors or Visual Colors technique. Each change replaces an existing color in the Custom Colors list.
- **Numeric colors** — You can enter either the Hue, Sat (Saturation), and Lum (Luminosity) values, or the Red–Green–Blue (RGB) values in the fields provided. The specified color appears in the Color|Solid box as you change each number.
- **Visual colors** — You can visually choose the Hue and Sat values by pressing and holding the mouse `SELECT` button and moving the mouse pointer in the large rainbow square to the color you want. You can visually choose the Lum value by moving the luminosity slider (at right) up or down. For each choice, the specified color appears in the Color|Solid box as you move the mouse.

Clicking the **OK** button assigns the color that currently appears in the Color|Solid box of the Windows dialog to the dynamic color number specified in your `SYSTEM-DIALOG COLOR` statement.

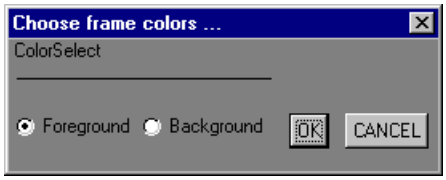
Color dialog box example

The `i-cdial1.p` procedure opens the dialog box that allows you to change its own foreground or background colors.

i-cdial1.p

```
DEFINE VARIABLE BackColor   AS INTEGER NO-UNDO INITIAL 17.
DEFINE VARIABLE ColorSelect AS INTEGER NO-UNDO INITIAL 16
  VIEW-AS RADIO-SET RADIO-BUTTONS "Foreground", 16, "Background", 17
  HORIZONTAL.
DEFINE VARIABLE FrontColor  AS INTEGER NO-UNDO INITIAL 16.
DEFINE VARIABLE status-ok   AS LOGICAL NO-UNDO.

DEFINE BUTTON bCANCEL       LABEL "CANCEL".
DEFINE BUTTON bOK           LABEL "OK".

IF COLOR-TABLE:NUM-ENTRIES < 18 THEN
  COLOR-TABLE:NUM-ENTRIES = 18.

status-ok = COLOR-TABLE:SET-DYNAMIC(16, TRUE).
IF NOT status-ok THEN DO:
  MESSAGE "Cannot make color 16 dynamic.".
  RETURN.
END.

status-ok = COLOR-TABLE:SET-DYNAMIC(17, TRUE).
IF NOT status-ok THEN DO:
  MESSAGE "Cannot make color 17 dynamic.".
  RETURN.
END.

COLOR-TABLE:SET-RGB-VALUE(16,RGB-VALUE(0,0,0)).
COLOR-TABLE:SET-RGB-VALUE(17,RGB-VALUE(128,128,128)).

FORM
  SKIP(0.5) SPACE(0.5) ColorSelect SPACE(2) bOK SPACE(2) bCANCEL
  SPACE(0.5) SKIP(0.5)
  WITH FRAME fColor TITLE "Choose frame colors ..." FGCOLOR FrontColor
    BGCOLOR BackColor VIEW-AS DIALOG-BOX.

ON CHOOSE OF bOK IN FRAME fColor DO:
  ASSIGN ColorSelect.
  SYSTEM-DIALOG COLOR ColorSelect.
END.

ON CHOOSE OF bCANCEL IN FRAME fColor STOP.

ENABLE ColorSelect bOK bCANCEL WITH FRAME fColor.

WAIT-FOR WINDOW-CLOSE OF FRAME fColor.
```

When you run this procedure on Windows, the following frame appears:



If you click **OK**, a color dialog box appears (see Figure 30: Windows system color dialog box on page 333) to assign a new system color to the specified color number. Clicking **CANCEL** terminates the procedure without any further color changes.

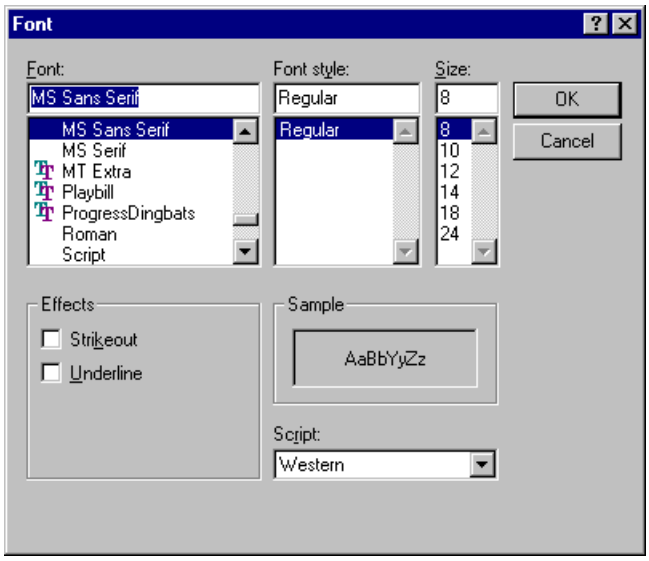
Saving a modified color

If you want to save any color definitions changed by the user, you can use the UPDATE option of the SYSTEM-DIALOG COLOR statement. This option sets a logical value to indicate whether the user has changed the specified color. If the value is TRUE, you can then save the new color definition using the PUT-KEY-VALUE statement.

Font dialog box

The following figure shows the native Windows system **Font** dialog box.

Figure 31: Windows Font dialog box



The **Windows** dialog box allows you to select a single font by name and specify a style, size, script, and optional cosmetic effect. A sample of each choice appears in the **Sample** box.

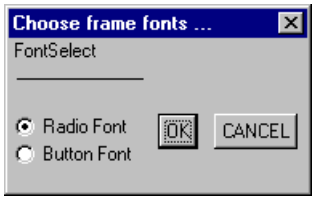
Font dialog box example

The i-fdial1.p procedure opens the dialog box that allows you to separately change the font of either its radio set or buttons to a custom font.

i-fdial1.p

```
IF FONT-TABLE:NUM-ENTRIES < 13 THEN
  FONT-TABLE:NUM-ENTRIES = 13.

DEFINE VARIABLE ButtonFont AS INTEGER NO-UNDO INITIAL 12.
DEFINE VARIABLE FontSelect AS INTEGER NO-UNDO INITIAL 11
  VIEW-AS RADIO-SET RADIO-BUTTONS "Radio Font", 11, "Button Font", 12
  FONT RadioFont.
DEFINE VARIABLE RadioFont  AS INTEGER NO-UNDO INITIAL 11.

DEFINE BUTTON bCANCEL LABEL "CANCEL" FONT ButtonFont.
DEFINE BUTTON bOK     LABEL "OK"     FONT ButtonFont.

FORM
  SKIP(0.5) SPACE(0.5) FontSelect SPACE(2) bOK SPACE(2) bCANCEL
  SPACE(0.5) SKIP(0.5)
  WITH FRAME fFont TITLE "Choose frame fonts ..." VIEW-AS DIALOG-BOX.

ON CHOOSE OF bOK IN FRAME fFont DO
  ASSIGN FontSelect.
  SYSTEM-DIALOG FONT FontSelect.
END.

ON CHOOSE OF bCANCEL IN FRAME fFont STOP.

ENABLE FontSelect bOK bCANCEL WITH FRAME fFont.

WAIT-FOR WINDOW-CLOSE OF FRAME fFont.
```

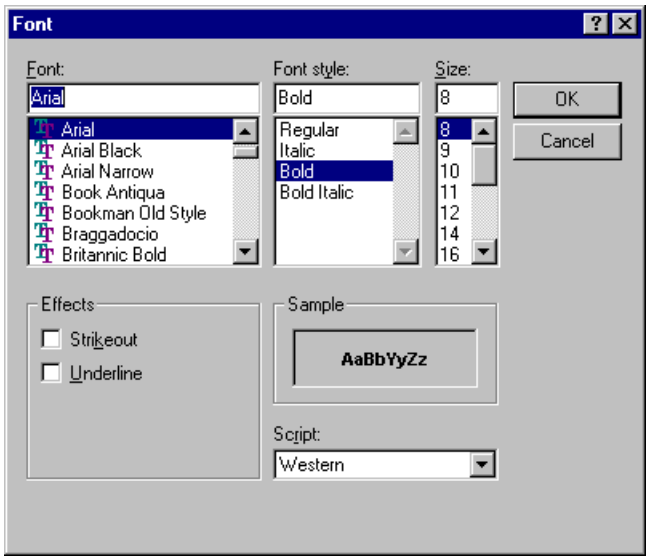
When you run this procedure in Windows, the following dialog box appears:



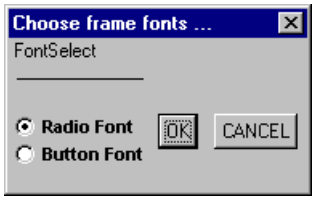
When you run the procedure, the radio set displays in the default MS Sans Serif font and buttons display in the default Courier New font. To change a font, select the **Radio Font** (font 11) or **Button Font** (font 12) radio item, then click **OK** to open the font dialog box shown in the following figure. Click **CANCEL** to terminate the procedure without any further font changes.

To change the radio-set font selected in the Figure 31: Windows Font dialog box on page 335 figure to bold 8-point Arial, set up the Font dialog as shown in the following figure.

Figure 32: Changing font 11 to bold 8-point Arial



After you click **OK**, the **Choose frame fonts** dialog box reappears with the radio set in the new font. For example:



Saving a modified font

If you want to save any font definitions changed by the user, you can use the `UPDATE` option of the `SYSTEM-DIALOG FONT` statement. This option sets a logical value to indicate whether the user has changed the specified font. If the value is `TRUE`, you can then save the new font definition using the `PUT-KEY-VALUE` statement.

Accessing the current color and font tables

In graphical interfaces, you can get information about colors and fonts defined for the current environment by querying attributes and methods of the `COLOR-TABLE` and `FONT-TABLE` system handles. You can also change the color configuration of the current environment using additional methods of the `COLOR-TABLE` handle.

COLOR-TABLE handle

The `NUM-ENTRIES` attribute sets and returns the number of colors available in the color table. The following methods allow you to read existing color definitions and to set new definitions for dynamic colors:

- The GET-RED-VALUE(*n*), GET-GREEN-VALUE(*n*), and GET-BLUE-VALUE(*n*) methods let you read the red, blue, and green values for a specified color number. This example gets the red value of the 16th color in the color table:

```
DEFINE VARIABLE red-val AS INTEGER NO-UNDO.
red-val = COLOR-TABLE:GET-RED-VALUE(16)
```

- You can determine whether a color is dynamic by reading the GET-DYNAMIC(*n*) method. You can make a color dynamic or nondynamic by using the SET-DYNAMIC(*n, logical*) method. This example makes color 16 dynamic if it is not already set:

```
IF NOT COLOR-TABLE:GET-DYNAMIC(16) THEN
  COLOR-TABLE:SET-DYNAMIC(16, TRUE).
```

- You can change the red, blue, and green values of a dynamic color by using the SET-RED-VALUE(*n, integer*), SET-GREEN-VALUE(*n, integer*), and SET-BLUE-VALUE(*n, integer*) methods. Note that these methods change the effective color values of a color number in the current color table. Thus, every visible widget set to the specified color number changes to the corresponding colors immediately. However, this does **not** change the color definitions originally specified in the current environment. If you restart the application with the current environment, the original colors reappear. This example sets color 16 to a blue value of 192:

```
COLOR-TABLE:SET-BLUE-VALUE(16, 192).
```

- There is an efficient alternative to calling the SET-RED-VALUE(*n, integer*), SET-GREEN-VALUE(*n, integer*), and SET-BLUE-VALUE(*n, integer*) methods to change the RGB values of a dynamic color. You can specify the SET-RGB-VALUE( ) method to substitute for all three of these methods. You can determine a combined RGB value using the RGB-VALUE function or by accessing a color property from an ActiveX control. For example:

```
COLOR-TABLE:SET-RGB-VALUE (16, RGB-VALUE (128, 0, 128)).
```

  The GET-RGB-VALUE( ) method returns an INTEGER that represents a combination of the red, green, and blue values associated with the nth entry. This combined RGB value is most useful for setting colors in an ActiveX control.

For more information on the COLOR-TABLE handle, see *ABL Reference*.

FONT-TABLE handle

The NUM-ENTRIES attribute sets and returns the number of fonts available in the font table. The FONT-TABLE methods return information about font sizes. Each method takes an optional font number as an argument:

- GET-TEXT-HEIGHT-CHARS( ) and GET-TEXT-HEIGHT-PIXELS( ) return the height of the default font in either character units or pixels. You can pass a font number to these methods to get the height of a

specific font. This example looks for the first font from the start of the font table with a height of at least two character units:

```
DEFINE VARIABLE font-number AS INTEGER NO-UNDO.
DEFINE VARIABLE max-count   AS INTEGER NO-UNDO.

max-count = FONT-TABLE:NUM-ENTRIES + 1.
REPEAT font-number = 1 to max-count:
  IF font-number = max-count THEN LEAVE.
  IF FONT-TABLE:GET-TEXT-HEIGHT-CHARS(font-number) >= 2 THEN LEAVE.
END.
```

- `GET-TEXT-WIDTH-CHARS( string )` and `GET-TEXT-WIDTH-PIXELS(string)` return the width of the passed *string* expression in the default font in either character units or pixels. You can optionally pass a font number to these methods to get the width of the string in a specific font. This example tests whether a title for a new window will fit based on the font of the current window:

```
DEFINE VARIABLE font-number AS INTEGER   NO-UNDO.
DEFINE VARIABLE in-title    AS CHARACTER NO-UNDO.
font-number = CURRENT-WINDOW:FONT.
SET in-title.
IF FONT-TABLE:GET-TEXT-WIDTH-CHARS(font-number, in-title) > 80 THEN
  MESSAGE "Title" in-title "cannot fit in new window.".
```

Retrieving and changing color and font definitions

In graphical interfaces, you can retrieve and change the color and font definitions for your current environment using the GET-KEY-VALUE and PUT-KEY-VALUE statements. These statements read and write to the current environment.

Changing resource definitions

The GET-KEY-VALUE statement can read and the PUT-KEY-VALUE statement can change the definition of any environment resource, including the definitions of colors and fonts stored in the current environment. However, these statements by themselves do not affect the current OpenEdge environment and its color and font tables. To have any definitions that are created using the PUT-KEY-VALUE statement take effect, you must replace the current environment by reloading the current environment (see Managing application environments on page 340).

Portable color and font definitions

The portable and most effective way to change color and font definitions is to first change the definitions in the color and font tables, then use the PUT-KEY-VALUE statement to write the new definitions from the modified tables. You can affect the color and font tables in the current environment using the COLOR-TABLE and FONT-TABLE handles (see Accessing the current color and font tables on page 337), and you can allow the user to affect the current color and font tables using the SYSTEM-DIALOG COLOR and SYSTEM-DIALOG FONT statements (see Allowing the user to change colors and fonts on page 332).

Using GET-KEY-VALUE and PUT-KEY-VALUE

The GET-KEY-VALUE and PUT-KEY-VALUE statements allow you to read or write a specified value for any resource by accessing the registry or an initialization file. The registry consists of sections called keys and subkeys arranged in a hierarchy. Keys and subkeys contain value entries, each of which consists of a value name and value data. Initialization files, by contrast, consist of a single level of sections. Sections contain entries, each of which consists of a name, an equals sign (=), and a value.

For example, to retrieve the Windows definition for font 8 from the current environment, which might be the registry or an initialization file, use a statement that returns the initial environment definition for font 8 in the FontString variable. For example:

```
DEFINE VARIABLE FontString AS CHARACTER NO-UNDO FORMAT "x(128)".
GET-KEY-VALUE SECTION "Fonts" KEY "Font8" VALUE FontString.
```

To specify "Times New Roman" as the new definition for font 8 in Windows, you might enter this statement, which sets the font8 parameter in the current environment:

```
PUT-KEY-VALUE SECTION "fonts" KEY "font8" VALUE "Times New Roman".
```

Writing portable color and font definitions

To write portable color and font definitions directly from the current color and font tables, use the PUT-KEY-VALUE statement with the COLOR or FONT option. For example, if you allow the user to change color 8 during a session through the color common dialog (SYSTEM-DIALOG COLOR statement), you can save the new color definition in the current environment using this statement:

```
PUT-KEY-VALUE COLOR 8.
```

You can save all current color definitions from the color table using this statement:

```
PUT-KEY-VALUE COLOR ALL.
```

For more information on these statements, see *ABL Reference*.

Managing application environments

For some graphical applications, you might want to replace whole OpenEdge environments. This applies especially to applications that build other applications, where you want to provide the ability to test run each application in a separate environment. The OpenEdge AppBuilder is an example of such an application. You might also want to write a single application that has greater flexibility to change its own environment (for example, changing the number of color definitions available to the application).

Understanding environment management

When OpenEdge starts up, it loads a default environment from the registry (Windows only) or from an initialization file. This default environment is the initial current environment. The initial current environment provides resources to the default window and any windows subsequently created in that environment. At any point, you can replace the current environment by loading and making a specified environment current.

Using environment management statements

To replace the current environment with another application environment, ABL provides these statements:

- **LOAD** — Loads a previously defined application environment into ABL memory, or creates and loads an empty application environment to be defined. Generally, an application environment resides in the Registry or in an initialization file with a meaningful filename. You can load as many application environments as memory allows, but only one environment can be current at a time.
- **USE** — Makes a previously loaded environment available as the current environment. The new current environment has no effect on the default window or any previously created windows. The resources of the new current environment are available only for windows created after the new environment is made current.
- **UNLOAD** — Removes a loaded environment from ABL memory. Environments take up very little memory and generally do not have to be unloaded. However, if you are writing an application builder, this statement provides good housekeeping when you save and remove an application from memory. This allows you to remove all memory associated with a purged application.

  Note that you must delete all windows created in an environment before you can unload that environment, whether or not it is the current environment. If the unloaded environment is the current environment, you must set a new current environment with the USE statement before continuing operation.

For more information on these statements, see their reference entries in *ABL Reference*.

Managing multiple environments

If you are writing an application builder, you might want to maintain separate environments for each application.

To separate application environments:

1. Select an existing application or create a new one.
2. Load the existing application environment, or create and load a new one using the LOAD statement.
3. Execute the USE statement to make the selected application environment current.
4. If the application exists, run it according to user input. If the application is new, do the following:
   a) Allocate the number of available color and font table entries for the new application by setting the NUM-ENTRIES attribute of the COLOR-TABLE and FONT-TABLE handles.
   b) Set internal colors and fonts from user input using the SYSTEM-DIALOG COLOR and FONT statements, or set internal colors using the COLOR-TABLE handle.
   c) Create application widgets and code from user input using the current color and font definitions.
5. Test and revise application resources and widgets according to user input.
6. Repeat Steps 1 through 5 for as many applications as you need to work on simultaneously.
7. When the application terminates, do the following:

a) Save the color and font table entries to the current environment using the PUT-KEY-VALUE statement with the COLOR and FONT options.

b) Save the application code to procedure files.

c) Delete all widgets for the application from ABL memory using the DELETE WIDGET statement.

d) Remove the current application environment from ABL memory using the UNLOAD statement.

8. Execute the USE statement according to user input to set a new current environment, then repeat Steps 4 through 8 until all work is done.

Helpful hints for environment management

Many variations in the typical scenario are possible, but this is a summary of important tasks to consider in any application that uses multiple environments:

- Before defining colors and fonts for immediate use at runtime, set or adjust the size of the current color and font tables as required using the NUM-ENTRIES attribute of the COLOR-TABLE and FONT-TABLE handles.
- Use the PUT-KEY-VALUE statement with the COLOR and FONT options to save newly created and newly modified color and font definitions from the current color and font tables to the current environment.
- Always set a current environment with the USE statement before or after executing the UNLOAD statement for the current environment, even if the current environment you set is the default environment. Otherwise, the AVM returns an error when you attempt to display or enable a widget.

Managing color display limitations

Although OpenEdge allows up to 256 colors to be defined in an environment, whether and how the widgets in each environment display these colors is system dependent. For example, if you have 256-color widgets displayed from three separate environments, and each environment defines an entirely different set of 256 colors, some of the widgets might not be displayed with the correct colors. If your display (driver and hardware) supports a maximum of 256 colors at one time, the 256-color widgets from only one of your environments can be displayed exactly as specified. All other widgets must be displayed incorrectly to allow the widgets from one environment to display correctly.

In general, no matter what your display limitations, OpenEdge tries to ensure that the current window (the window that has focus) displays correct colors at the expense of one or more other windows that are displayed, but do not have focus. You can also do the following to ensure that all of your widgets display with correct colors:

- Do not define more colors for all loaded and used environments than your display can simultaneously support.
- Hide any widgets, especially images, that you no longer need to display and whose color content exceeds the color capacity of your system.

9

Creating Reports

A database management system provides information that helps an organization run efficiently. A report—whether printed on paper, viewed on a workstation, or stored on media—is a common tool for distributing database information.

These topics introduce you to the ABL (Advanced Business Language) statements that you can use to generate both simple and complex reports.

For details, see the following topics:

- Report basics
- Designing an interface for viewing report data
- Converting widget values to report data
- Generating reports with control breaks and aggregates
- Generating reports from multiple tables
- Redirecting output
- Designing frames for reports
- Using the PUT statement
- Summary

Report basics

The event-driven model provides a highly flexible environment for interactive applications. Reports collect and organize static data without interaction from the user, so traditional top-down programming techniques work better for this task. Typically, you can integrate a report-generating procedure into an event-driven application using a trigger with a RUN statement attached to a button or menu command (covered later).

The components of a basic report procedure include:

- A special type of frame, called a down frame, that can hold more than one iteration of data
- Frame phrase options to format the frame and included widgets
- Text widgets
- Format phrase options to format individual widgets
- An iterating control block
- An output statement

Down frames

Like an interactive display, a frame is the data container for reports. However, unlike a display frame, a report usually contains more than one iteration of data. A frame that can contain more than one iteration of data is known as a *down frame*. (Sometimes a frame with a single iteration of data is called a *one-down frame*.)

You might have learned that a frame is a container for field-level widgets (data widgets, action widgets, graphic widgets). Actually, it's a bit more complicated than that. All the widgets included in a one-down frame are contained in a special widget called a *field-group widget*. The frame contains the field-group widget. When you display several iterations of data, the widgets of each iteration belong to a single field-group widget, and the frame contains the set of field group widgets.

The behind-the-scenes work of the field-group widget is rarely something that you need to track. The explanation above simply serves to explain the mechanics of a down frame in terms of the programming model.

The default frame of an iterating control block, like FOR EACH, is a down frame. You can also specify a down frame with the DOWN keyword of the frame phrase.

Add an integer before DOWN to specify the maximum number of iterations the frame can hold. For example:

```
DEFINE FRAME Frame1
  sports2000.Customer.Name
  WITH 12 DOWN CENTERED THREE-D.
```

To fit as many iterations in the frame as the screen can hold, **do not** add an integer before the DOWN keyword. For example:

```
DEFINE FRAME Frame1
  sports2000.Customer.Name
  WITH DOWN CENTERED THREE-D.
```

Text widgets

The text widget displays textual data without the decorations native to your windowing system. This means that data displays more compactly, so more iterations fit in a single display or printed page.

If you are working with fields or variables that use the fill-in field as the default data widget, then the USE-TEXT option of the frame phrase quickly converts the fill-in fields to text widgets. For example:

```
DEFINE FRAME Frame1
  sports2000.Customer.Name
  WITH DOWN USE-TEXT CENTERED THREE-D.
```

If the default data widget is anything other than a fill-in field or text widget, you need to use the VIEW-AS TEXT option of the format phrase on each widget to convert the widget to a text widget. For example:

```
DEFINE VARIABLE lBalDue AS LOGICAL NO-UNDO LABEL "Balance Due?"
  VIEW-AS TOGGLE-BOX.

DEFINE FRAME Frame1
  sports2000.Customer.Name  /* default is a fill-in field */
  lBalDue VIEW-AS TEXT  /* default is a toggle box */
  WITH DOWN USE-TEXT CENTERED THREE-D.
```

Remember that the default data representation for a field comes from the schema of the database, while the default for a variable comes from the DEFINE VARIABLE statement.

Control blocks and output statements

Compiling a report usually involves a straightforward process of moving through a set of table records, calculating data, and outputting results. A control block best handles the report-generating process. Most reports use the FOR EACH block because of its record-reading properties. For example:

```
FOR EACH Customer NO-LOCK WITH FRAME Frame1:
  lBalDue = IF Customer.Balance > 0 THEN TRUE ELSE FALSE.
  DISPLAY Customer.Name lBalDue.
END.
```

The default frame of an iterating control block is a down frame.

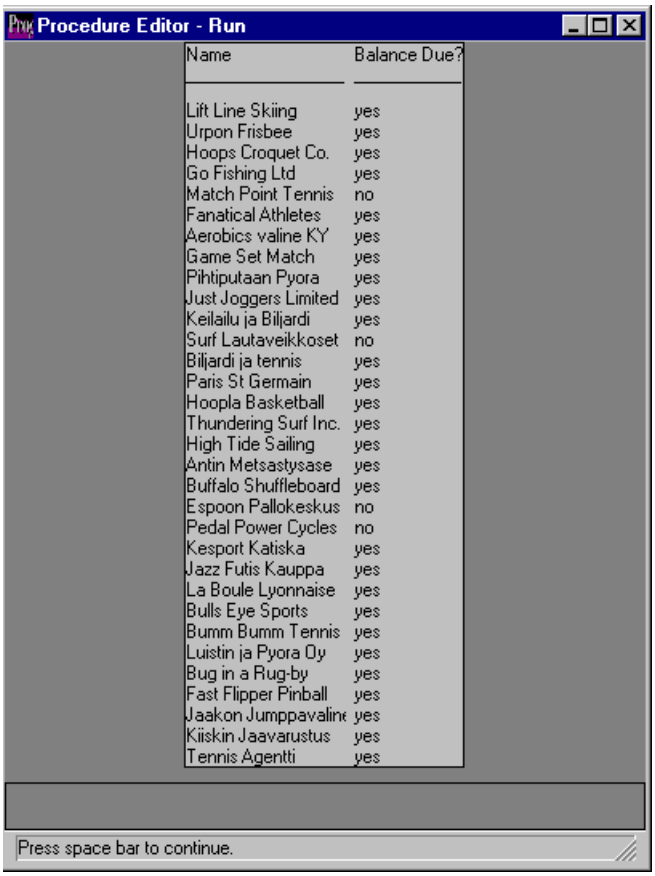
The DISPLAY statement is ABL's main programming statement for output. DISPLAY can output to printers and media as well as the screen. Later, you'll learn about another output statement and how to direct output.

Basic report demonstration

This section provides a basic report demonstration.

To see a demonstration of ABL behavior when displaying a report to your screen:

1. Open i-10-01.p and run it. The ABL Virtual Machine (AVM) creates a down frame that occupies all the vertical space it can and fills the frame with iterations of data. For example:



Notice the message at the bottom of the screen. the AVM pauses to allow you to view the first frame of data.

2. Press **SPACEBAR**. The AVM clears the down frame and fills it with a new set of iterations.
3. Press **END-ERROR** and then **SPACEBAR** to return to the Procedure Editor. Here is the code for this example:

i-10-01.p

```
DEFINE VARIABLE lBalDue AS LOGICAL NO-UNDO LABEL "Balance Due?"

        VIEW-AS TOGGLE-BOX.
      DEFINE FRAME Frame1
        sports2000.Customer.Name
/*1*/   lBalDue VIEW-AS TEXT
/*2*/   WITH DOWN USE-TEXT CENTERED THREE-D.
/*3*/ FOR EACH Customer NO-LOCK WITH FRAME Frame1:
        lBalDue = IF Customer.Balance > 0 THEN TRUE ELSE FALSE.
/*4*/   DISPLAY Customer.Name lBalDue.
      END.
```

Note: The `THREE-D` option is relevant only on a Windows client; it is ignored by a character client.

The following notes help to explain the code:

1. The default data widget for this `LOGICAL` variable is a toggle box, making the `VIEW-AS TEXT` phrase necessary to convert it to a text widget.
2. The frame phrase of the `DEFINE FRAME` statement contains two key options: `DOWN` and `USE-TEXT`. The `DOWN` keyword specifies a down frame that automatically expands to fit as many iterations as the screen can hold. Specifying an integer before `DOWN` sets the maximum number of iterations the frame can hold. `USE-TEXT` converts all fill-in fields into text widgets for a compact display.
3. Since the great majority of reports compile and manipulate table data, the `FOR EACH` statement is the most common control block used for report procedures. The frame phrase here replaces the default down frame with the named down frame defined earlier.
4. The `DISPLAY` statement is the most commonly used output statement. As you'll see later, `DISPLAY` can output to terminals, printers, or files.

The type of report display created by procedure `i-10-01.p` does not fit well with the event-driven model because the user:

- Does not control the display of data
- Can view data in only one direction
- Does not have an intuitive way to dismiss the report

Designing an interface for viewing report data on page 347 describes a technique for viewing reports that better suits the event-driven model.

Designing an interface for viewing report data

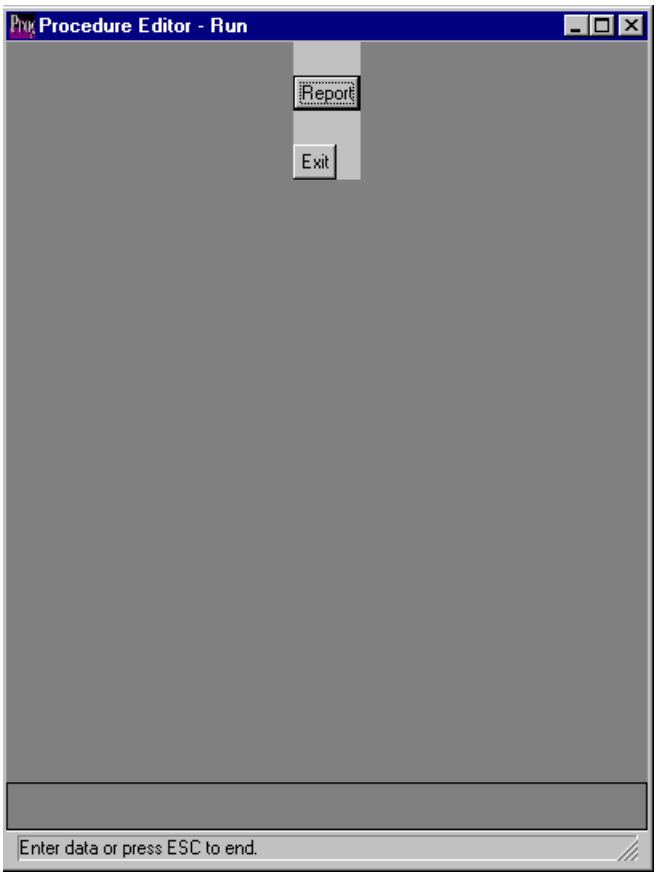
In contrast to the ABL-controlled interface you saw in the last example, you need the following to create a user-controlled report viewing interface:

1. A dialog box to contain the output, so report data does not require space on your main display.
2. An **OK** button in the dialog box to let the user dismiss the report.
3. A temporary file to contain the report data. Having the control block output to a temporary file eliminates the pausing behavior.
4. An editor widget with scrollbars for navigating the report.

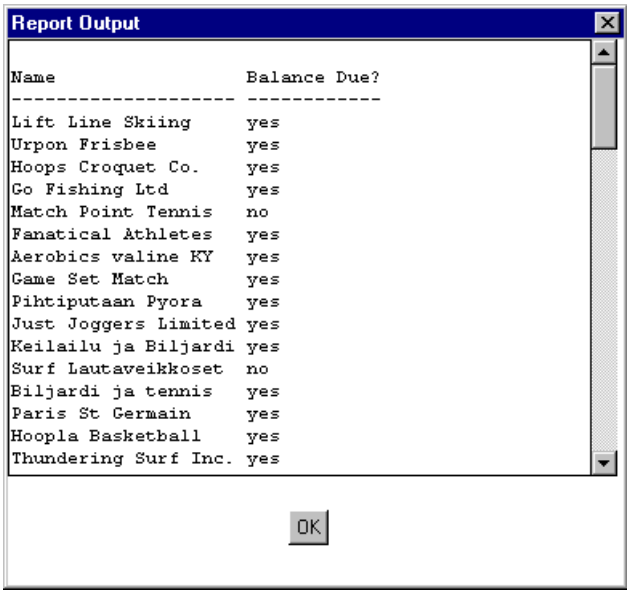
The next example creates a dialog box interface for the same report you saw in the last example. The notes that follow the exercise explain the new code techniques you need to make this interface work.

To create a dialog box interface for the report in Windows:

1. Open `i-10-02.p` and run it. The following screen appears:



2. Click **Report**. The **Report Output** dialog box appears:



The report data appears in an editor that you can scroll through.

3. Click **OK** to dismiss the dialog box.
4. Click **Exit** and then press **SPACEBAR** to return to the Procedure Editor.

Here is the code for this example:

i-10-02.p

```
 /**********  DEFINE WIDGETS  **********/
      DEFINE VARIABLE lBalDue    AS LOGICAL   NO-UNDO LABEL "Balance Due?"
        VIEW-AS TOGGLE-BOX.
      DEFINE VARIABLE lStat      AS LOGICAL   NO-UNDO.
/*1*/ DEFINE VARIABLE Rep-Editor AS CHARACTER NO-UNDO VIEW-AS EDITOR
        SCROLLBAR-VERTICAL SIZE 79 BY 16.

      DEFINE BUTTON b-exit LABEL "Exit".
      DEFINE BUTTON b-ok   LABEL "OK" AUTO-GO.
      DEFINE BUTTON b-rep  LABEL "Report".

      /**********  DEFINE FRAMES  **********/
      DEFINE FRAME Frame1
        SKIP(1)
        b-rep SKIP(1)
        b-exit
        WITH NO-BOX CENTERED THREE-D.
      DEFINE FRAME Dialog1
        Rep-Editor SKIP(1)
        b-ok TO 40 SKIP(1)
        WITH NO-LABELS VIEW-AS DIALOG-BOX SCROLLABLE.

      /**********  DEFINE TRIGGERS  **********/
      ON CHOOSE of b-rep DO:
/*2*/   OUTPUT TO "tut-temp.txt".
/*3*/   FOR EACH Customer WITH STREAM-IO:
          lBalDue = IF Customer.Balance > 0 THEN TRUE ELSE FALSE.
          DISPLAY Customer.Name lBalDue.
        END.
```

```
/*4*/   OUTPUT CLOSE.
/*5*/   ASSIGN
          Rep-Editor:READ-ONLY IN FRAME Dialog1 = TRUE
          Rep-Editor:SENSITIVE IN FRAME Dialog1 = TRUE
          FRAME Dialog1:TITLE                   = "Report Output"
          lStat = Rep-Editor:READ-FILE("tut-temp.txt") IN FRAME Dialog1.
/*6*/   IF lStat THEN DO:
          ENABLE Rep-Editor b-ok WITH FRAME Dialog1.
          WAIT-FOR GO OF FRAME Dialog1.
/*7*/     HIDE FRAME Dialog1.
        END.
      END.

      /**********  MAIN LOGIC  **********/
/*8*/ ASSIGN Rep-Editor:FONT = 3.
      ENABLE ALL WITH FRAME Frame1.
      WAIT-FOR CHOOSE OF b-exit.
```

These notes explain the new code techniques used in `i-10-02.p`:

1. This editor is for the report data.
2. The `OUTPUT TO` statement is your tool for directing output. This example directs output to the named file. Directing output is more fully discussed in a later section.
3. This `FOR EACH` uses the default down frame, and the `STREAM-IO` option reduces all widgets to textual data without decorations. Using the `STREAM-IO` option is a requirement when outputting to any destination other than the screen.

4. Once you direct output to a destination, you need to close the output stream to return output to the default destination, which is the screen.
5. This ASSIGN statement manipulates three editor attributes and one method to set up the dialog box. The READ-ONLY attribute prevents changes to the report content. The SENSITIVE attribute enables the editor, making the scrolling functions available. The TITLE attribute contains the dialog box title text. The READ-FILE method takes a valid filename and returns YES or NO to indicate if the AVM successfully read the file contents into the editor.
6. You only want to bring up the dialog box if the logical variable lStat equals YES, which means that the AVM successfully read the file into the editor.
7. If you use a WAIT-FOR statement in a dialog box, you must use the HIDE statement to dismiss the dialog box.
8. On all platforms that OpenEdge supports, the output font designated by 3 is always a monospaced font, which are frequently used in printed reports. Assigning this font makes the editor contents mimic a printed report.

Converting widget values to report data

When reporting, you want to output data, but not the widget that represents that data. Of the data widgets, you can directly output the data of these widgets to all devices: fill-in fields, text widgets, and editors. You can output these data widgets to the screen only: toggle box, radio set, slider, combo box, and selection list.

In the last section, you learned about the USE-TEXT and VIEW-AS text options to convert other widgets to text widgets. In this section, you will learn about two more important techniques for handling widgets:

- Using the STREAM-IO option of the frame phrase to convert all data widgets in a frame to text widgets
- Using the editor for formatting long text strings as blocks of text

Printing reports and the STREAM-IO option

For toggle boxes, radio sets, sliders, combo boxes, and selection lists, trying to output the widget to any other device besides the screen yields nothing. Neither the widget nor the value it contains appears in files or printed reports.

Rather than tack a VIEW-AS TEXT phrase onto every widget, you can use the STREAM-IO option of the frame phrase to reduce every widget to textual data. Here is an example:

```
DEFINE VARIABLE lBalDue AS LOGICAL NO-UNDO LABEL "Balance Due?"
  VIEW-AS TOGGLE-BOX.

DEFINE FRAME Frame1
  sports2000.Customer.Name
  lBalDue
  WITH DOWN STREAM-IO CENTERED THREE-D.
```

You must use the STREAM-IO option on every output frame intended for a destination other than the screen.

Although primarily a tool for sending data to files and reports, the STREAM-IO option also reduces all widgets to textual data for screen output.

To see the effects of the STREAM-IO option on screen data:

1. Open `i-10-03.p` and run it. This report is the same as the one from the previous exercise, but without the special interface.
2. Press **END-ERROR** and then **SPACEBAR** to return to the Procedure Editor.
3. Remove `STREAM-IO` from the frame phrase and run the procedure again.

   Notice that the `lBalDue` variable displays as a disabled toggle box in the iterations.
4. Press **END-ERROR** and then **SPACEBAR** to return to the Procedure Editor.

If you scroll through the text of the procedure, you can see that the `STREAM-IO` option on the frame phrase of the iterating block is the only change.

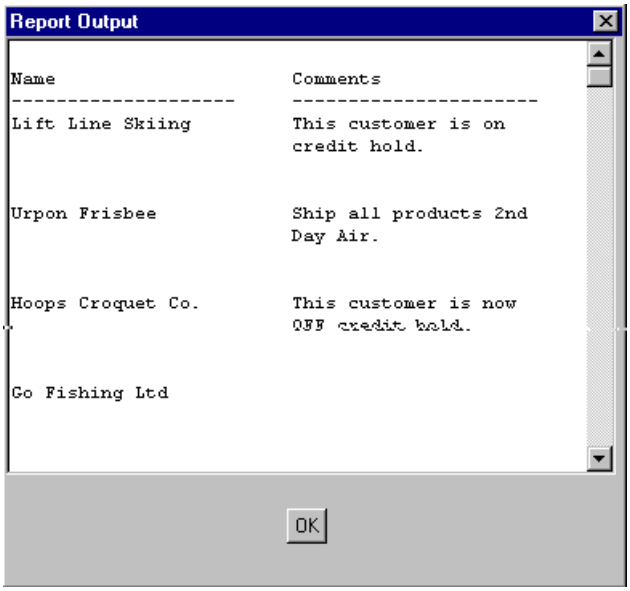
Formatting long text strings

The remaining problem for simple output is how to handle long text strings. A text widget that contains a long string may cause formatting problems such as skipping to a new line or truncating. For screen design, you saw earlier that the editor widget is a good choice for holding long text strings. The editor is also the solution for reports. With the editor, you can output text in blocks.

For example, the Comments field in the **Customer** table has a format of `x(60)`. Reserving 60 spaces takes up most of a line in a standard printed report. You can override this default behavior by specifying a `VIEW-AS EDITOR` phrase for the Comments field. When you output a field or variable value as an editor widget, the AVM uses the `INNER-CHARS` attribute of the editor to format and wrap the text into a block. The `INNER-LINES` syntax sets the minimum number of lines the block occupies. If there is enough data to fill more lines than specified by `INNER-LINES`, then the AVM provides the extra room.

To see an example of editor output in reports:

1. Open `i-10-04.p` and run it.
2. Click **Report**. The **Report Output** dialog box appears:



   As you scroll through the editor, notice the blocks of **Comments** text.
3. Click **OK**, then **Exit**, and then press **SPACEBAR** to return to the Procedure Editor.

Here is the code for this procedure:

i-10-04.p

```
/* {i-10-in.i} Common Interface Setup Code */

/**********  DEFINE TRIGGERS  **********/
ON CHOOSE of b-rep DO:
  OUTPUT TO "tut-temp.txt".
  FOR EACH Customer NO-LOCK WITH STREAM-IO:
    DISPLAY Customer.Name SPACE(5) Customer.Comments
      VIEW-AS EDITOR INNER-LINES 3 INNER-CHARS 20.
  END.
  OUTPUT CLOSE.

  ASSIGN
    Rep-Editor:READ-ONLY IN FRAME Dialog1 = TRUE
    Rep-Editor:SENSITIVE IN FRAME Dialog1 = TRUE
    FRAME Dialog1:TITLE                   = "Report Output"
    lStat = Rep-Editor:READ-FILE("tut-temp.txt") IN FRAME Dialog1.

  IF lStat THEN DO:
    ENABLE Rep-Editor b-ok WITH FRAME Dialog1.
    WAIT-FOR GO OF FRAME Dialog1.
    HIDE FRAME Dialog1.
  END.
END.

/**********  MAIN LOGIC  **********/
ENABLE ALL WITH FRAME Frame1.
WAIT-FOR CHOOSE OF b-exit.
```

Note:

`INNER-LINES` reserves at least three lines per iteration of the report. `INNER-CHARS` sets the length of the text block at 25 characters.

`INNER-LINES` reserves at least three lines per iteration of the report. `INNER-CHARS` sets the length of the text block at 20 characters.

This section completes the basic discussion on creating a reporting procedure. The next few sections concentrate on the techniques for populating reports with more complex data.

Generating reports with control breaks and aggregates

You can sort data using the `BY` option of the record phrase. For example, this block header in a report procedure sorts the **Customer** records by the `SalesRep` code:

```
FOR EACH Customer NO-LOCK BY Customer.SalesRep:
```

A *control break* separates sorted data into meaningful groups. Breaking sorted data by `SalesRep` separates each sales rep's customers into a separate *break group*. Break groups allow you to calculate summary data on subsets of data. For example, you could total the number of customers that each sales rep serves.

The BREAK option of the record phrase sets up your data for control breaks, as shown in this block header statement:

```
FOR EACH Customer NO-LOCK BREAK BY Customer.SalesRep:
```

Once you've set up the control break, you can use an aggregate phrase on an output statement item to calculate summary information. This is the syntax of an aggregate phrase:

Syntax

```
[ aggregate-option ] ... [ BY break-group ] ...
```

The following table lists the aggregate options.

Table 44: Aggregate options

Aggregate option	Description
AVERAGE	Calculates the average of all the expression's values, and calculates the overall average for the break groups
SUB-AVERAGE	Calculates the average of all the expression's values, but does not calculate the overall average
COUNT	Counts the number of items in the break group, and calculates the overall count of the break groups
SUB-COUNT	Counts the number of items in the break group, but does not calculate the overall count
MAXIMUM	Finds the maximum value of the expression in the break group, and finds the overall maximum of the break groups
SUB-MAXIMUM	Finds the maximum value of the expression in the break group, but does not find the overall maximum
MINIMUM	Finds the minimum value of the expression in the break group, and finds the overall minimum of the break groups
SUB-MINIMUM	Finds the minimum value of the expression in the break group, but does not find the overall minimum
TOTAL	Calculates the total of all the expression's values, and calculates the overall total for the break group.
SUB-TOTAL	Calculates the total of all the expression's values, but does not calculate the overall total for the break groups

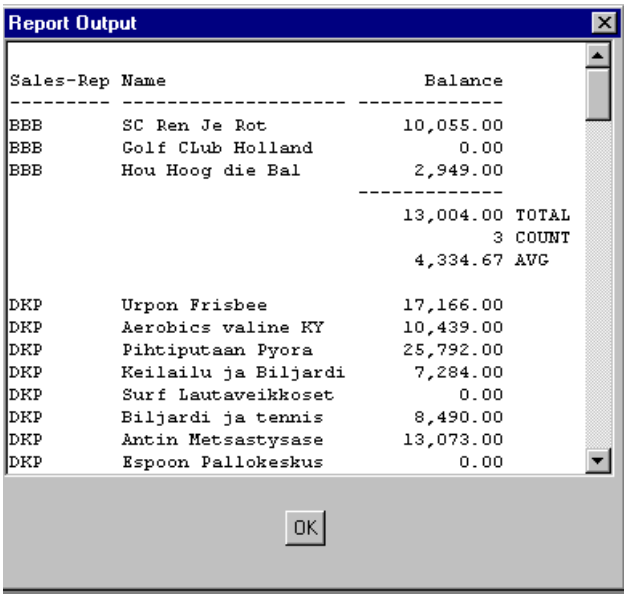
In the example below, SalesRep is the control break, Balance is the expression, and the aggregate phrase with the TOTAL option appears in parenthesis immediately after the expression in the DISPLAY statement:

```
FOR EACH Customer NO-LOCK
  BREAK BY Customer.SalesRep:
  DISPLAY Customer.Balance (TOTAL BY Customer.SalesRep).
END.
```

The TOTAL aggregate option totals the balances for each break group. Each time the value of SalesRep changes in the sorted data, the AVM creates another break group, and the TOTAL BY aggregate function calculates and outputs the total of customer balances for that sales rep's customers. Finally, at the end of the report, the TOTAL option calculates and outputs the total balances of all the break groups.

To see an example of control breaks and aggregate values:

1. Open i-10-05.p and run it.
2. Click **Report**. The **Report Output** dialog box appears:



   The members of each break group all have the same SalesRep value. Notice that the report separates each break group with white space and outputs the result of the aggregate phrase options immediately following the break groups.

   If you scroll to the bottom of the report, you can see the aggregate values for all the break groups.

3. Click **OK**, then **Exit**, and then press **SPACEBAR** to return to the Procedure Editor.

Here is the code for this procedure:

i-10-05.p

```
    /* {i-10-in.i} Common Interface Setup Code */

    /**********  DEFINE TRIGGERS  **********/
    ON CHOOSE OF b-rep DO:
      OUTPUT TO "tut-temp.txt".
/*1*/ FOR EACH Customer NO-LOCK
        BREAK BY Customer.SalesRep WITH STREAM-IO:
        DISPLAY Customer.SalesRep Customer.Name
/*2*/     Customer.Balance (COUNT TOTAL AVERAGE BY Customer.SalesRep).
      END.
      OUTPUT CLOSE.

      ASSIGN
        Rep-Editor:READ-ONLY IN FRAME dialog1 = TRUE
        Rep-Editor:SENSITIVE IN FRAME Dialog1 = TRUE
        FRAME dialog1:TITLE                   = "Report Output"
        lStat = Rep-Editor:READ-FILE("tut-temp.txt") IN FRAME Dialog1.

      IF lStat THEN DO:
        ENABLE Rep-Editor b-ok WITH FRAME Dialog1.
        WAIT-FOR GO OF FRAME Dialog1.
        HIDE FRAME Dialog1.
      END.
    END. /* ON CHOOSE OF b-rep */

    /**********  MAIN LOGIC  **********/
    ENABLE ALL WITH FRAME Frame1.
    WAIT-FOR CHOOSE OF b-exit.
```

Note:

The BREAK option of the record phrase sets up the control break.

The aggregate phrase after the Balance expression contains three aggregate options.

Generating reports from multiple tables

You have already learned all the language syntax necessary to:

- Sort information
- Relate information from different tables
- Derive new data with calculations and aggregate options

The reports you have seen so far display data from a single table (the **Customer** table) and demonstrate one or two reporting features. Real world reports, however, frequently do a little bit of everything as they strive to organize lots of information and calculate useful summary information.

For example, suppose the All Around Sports accounting department needs to analyze delinquent accounts. To do their jobs, the accountants need information on customers, orders, and inventory. To get all the data they need, the report requires information from four tables: the **Customer**, **Order**, **OrderLine**, and **Item** tables. The rest of this section builds a report that gathers and organizes information from these four tables and performs some summary calculations on the data.

Reporting information from one table

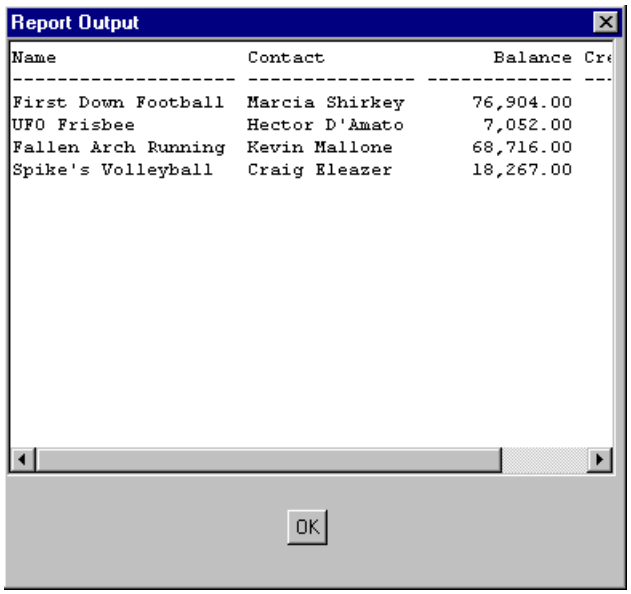
First, the accounting department needs to sort out which customers have exceeded their credit limits or those who are close to exceeding the limits. All Around Sports wants Accounting to look at all customers that have balances equal to or greater than 85 percent of their credit limits. All of this information is in the **Customer** table.

A single FOR EACH with a WHERE clause is what you need to gather this information. The following code fragment from i-10-06.p shows this structure.

i-10-06.p

```
...
FOR EACH Customer NO-LOCK
  WHERE Customer.Balance >= (Customer.CreditLimit * .85) WITH STREAM-IO:
  DISPLAY
    Customer.Name FORMAT "x(20)"
    Customer.Contact FORMAT "x(15)"
    Customer.Balance
    Customer.CreditLimit
    WITH NO-BOX.
END.
...
```

The WHERE clause selects just the customers over the 85 percent threshold, and the DISPLAY statement outputs the key information about these customers. Running this procedure yields the following output:



Reporting information from two tables

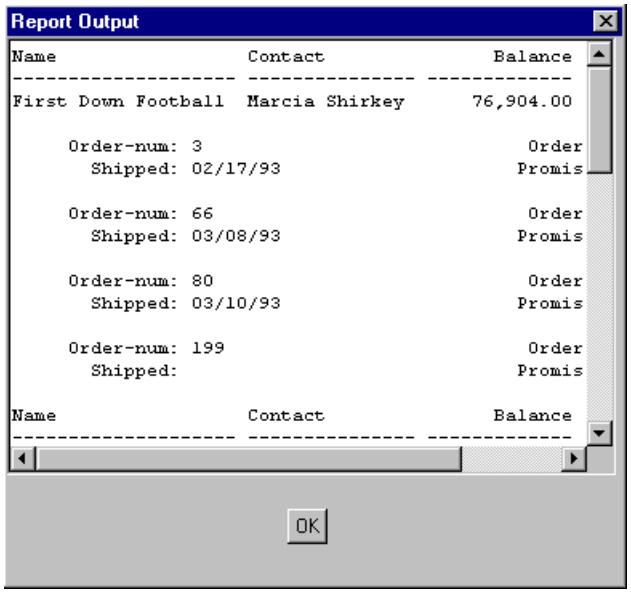
The next step is to look at the orders for each selected customer. This information is in the **Order** table. Another FOR EACH inside the first accomplishes this task. For each iteration of the outer block, the inner block finds all the related information. The WHERE clause establishes the relationship between the two tables.

The code from i-10-07.p shows the nested FOR EACH blocks.

i-10-07.p

```
...
FOR EACH Customer NO-LOCK
  WHERE Customer.Balance >= (Customer.CreditLimit * .85) WITH STREAM-IO:
  DISPLAY
    Customer.Name FORMAT "x(20)"
    Customer.Contact FORMAT "x(15)"
    Customer.Balance
    Customer.CreditLimit WITH NO-BOX.

  FOR EACH Order
    NO-LOCK WHERE Order.CustNum = Customer.CustNum WITH STREAM-IO:
    DISPLAY Order.OrderNum Order.OrderDate Order.ShipDate Order.PromiseDate
       SKIP(1) WITH 2 COLUMNS.
  END.
END.
...
```

The following output shows how the related information ends up grouped together:



Notice that for a single iteration of the **Customer** data, there may be several iterations of **Order** data. Earlier you learned that the default frame of an iterating control block is a down frame. When you nest control blocks, only the innermost block uses a down frame. The other blocks execute one iteration at a time. This default behavior lets the related information group together naturally with the iteration of the blocks.

Reporting information from multiple tables

Each order has one or many order lines, each of which relates to a single inventory item. Accounting needs to pull information from the **OrderLine** and **Item** tables to get specific information about what was ordered. One more `FOR EACH` block, inside the existing block, completes the structure.

Once the procedure gathers all the information, accounting needs the procedure to calculate total prices. Nested `FOR EACH` blocks create sorted groups of information similar to break groups. Therefore, you can use the aggregate phrase to calculate total prices.

The code fragment from `i-10-08.p` shows the completed structure of nested `FOR EACH` blocks.

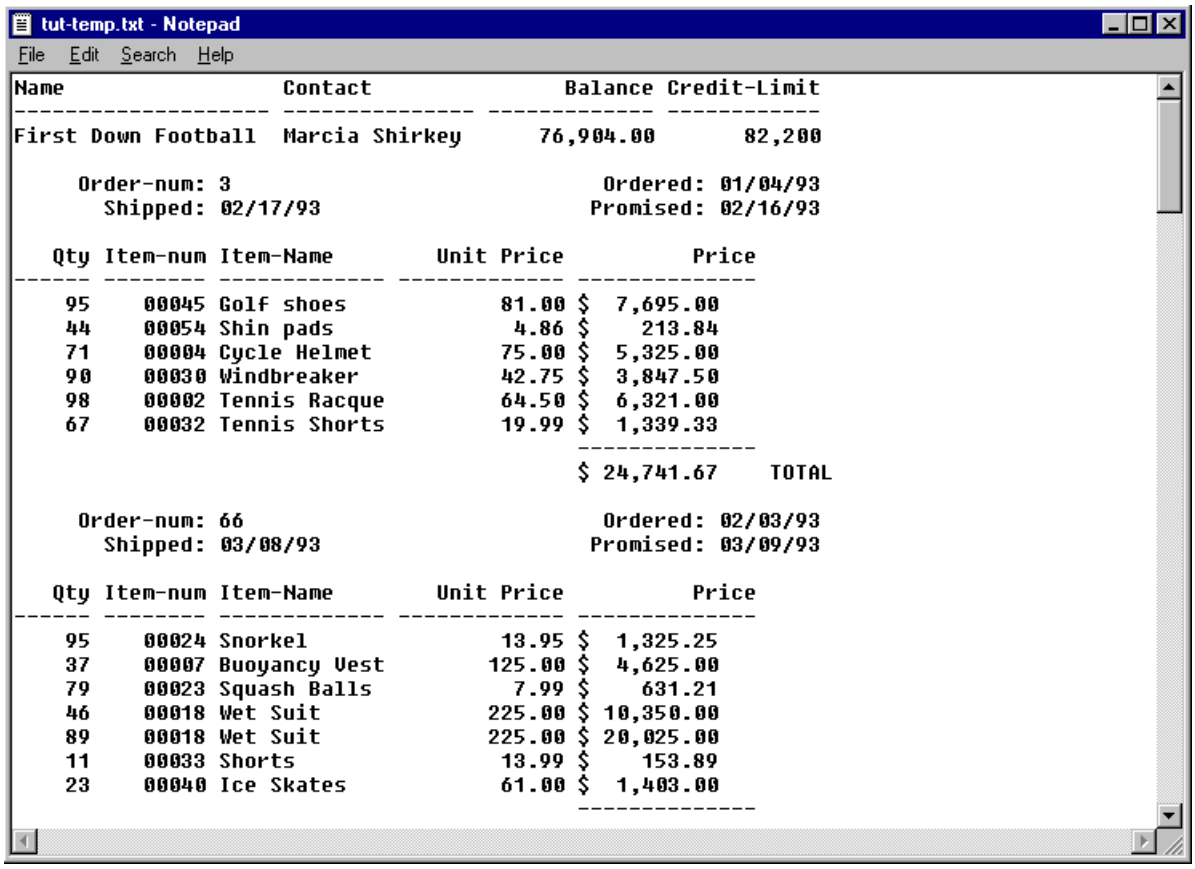
i-10-08.p

```
      . . .
      FOR EACH Customer NO-LOCK
        WHERE Customer.Balance >= (Customer.CreditLimit * .85)
        WITH STREAM-IO:
        DISPLAY Customer.Name FORMAT "x(20)" Customer.Contact FORMAT "x(15)"
          Customer.Balance Customer.CreditLimit WITH NO-BOX.

/*1*/   FOR EACH Order NO-LOCK WHERE Order.CustNum = Customer.Custum
          WITH STREAM-IO:
          DISPLAY Order.OrderNum Order.OrderDate Order.ShipDate
            Order.PromiseDate SKIP(1) WITH 2 COLUMNS.

          FOR EACH OrderLine NO-LOCK
            WHERE OrderLine.OrderNum = Order.OrderNum WITH STREAM-IO:
/*2*/       FIND Item NO-LOCK WHERE Item.ItemNum = OrderLine.ItemNum.
            DISPLAY OrderLine.Qty OrderLine.ItemNum
              Item.ItemName FORMAT "x(13)"
              Item.Price LABEL "Unit Price"
/*3*/           Item.Price * Qty (TOTAL) LABEL "Price"
/*4*/             FORMAT "$zzz,zz9.99 CR" WITH NO-BOX.
          END./* FOR EACH OrderLine */
        END. /* FOR EACH Order */
      END. /* FOR EACH Customer */
      . . .
```

These notes explain the code highlights:

- The `WHERE` clause of the third `FOR EACH` relates the **OrderLine** table back to the **Order** table.
- For each **OrderLine**, there is a single Item record that contains information about the ordered item. A simple `FIND` statement retrieves this information.
- The report totals the result of an expression. Notice the absence of the `BY` break group syntax.
- The `FORMAT` option here specifies a fairly complex format string. The result displays a leading dollar sign ($), suppresses the leading zeroes (z), and displays the credit symbol (CR) when the result is a negative value.

The following shows the output of the final version of this report:



Redirecting output

A report procedure outputs to the screen by default and takes control of the application by building reports and pausing one screen at a time. Since this behavior takes control away from the user, you also learned to create a more user friendly interface for reports based on the `OUTPUT TO` statement. Essentially, report output gets redirected from the screen to a temporary file, and then the procedure reads the report results into an editor in a dialog box. This section discusses the `OUTPUT TO` statement and shows you how to send report output to text files and printers.

Other uses include sending reports to text tables or UNIX devices. See *ABL Reference* for more information about these uses of the `OUTPUT TO` statement.

OUTPUT TO and the default stream

A *stream* is a path for data movement. In this case, a stream is a path for output to a named destination. Every ABL procedure has one unnamed default output stream that writes data to the screen. The `OUTPUT TO` statement is your tool for redirecting the default stream to another destination. Once you use `OUTPUT TO` to change the output destination in a procedure, the output goes to that destination until you close it with the `OUTPUT CLOSE` statement, or until you name a new output destination.

This is a partial syntax for the `OUTPUT TO` statement:

Syntax

```
OUTPUT [ STREAM stream | STREAM-HANDLE handle ]
  TO { PRINTER | opsys-file | opsys-device }
  [ PAGED ]
```

Here is a partial syntax for the OUTPUT CLOSE statement:

Syntax

```
OUTPUT [ STREAM stream | STREAM-HANDLE handle ] CLOSE
```

The following table describes the OUTPUT statement syntax components.

Table 45: OUTPUT statement options

Component	Description
STREAM	Specifies a named output stream. Omit when using the default stream.
STREAM-HANDLE	Specifies the handle to a named output stream. Omit when using the default stream.
PRINTER	Specifies the default printer as set by your environment.
opsys-file	Specifies a legal file name enclosed in quotes for a text file.
opsys-device	Specifies a legal device name from your environment enclosed in quotes.
PAGED	Directs the AVM to break the output into pages. The AVM sends a page break control character after every 56 lines. Without this option, the AVM sends output in a continuous stream. This option is most often used with printed reports, especially where page headers and footers are used (covered later).

Directing output to a file

To direct output to a file, use the OUTPUT TO statement followed by the name of the destination file enclosed in quotes. The destination can include the path, or just the filename. If you do not provide a path, the AVM writes the file to the current directory. If the file does not exist, the AVM creates it. If the file does exist, the AVM overwrites its contents.

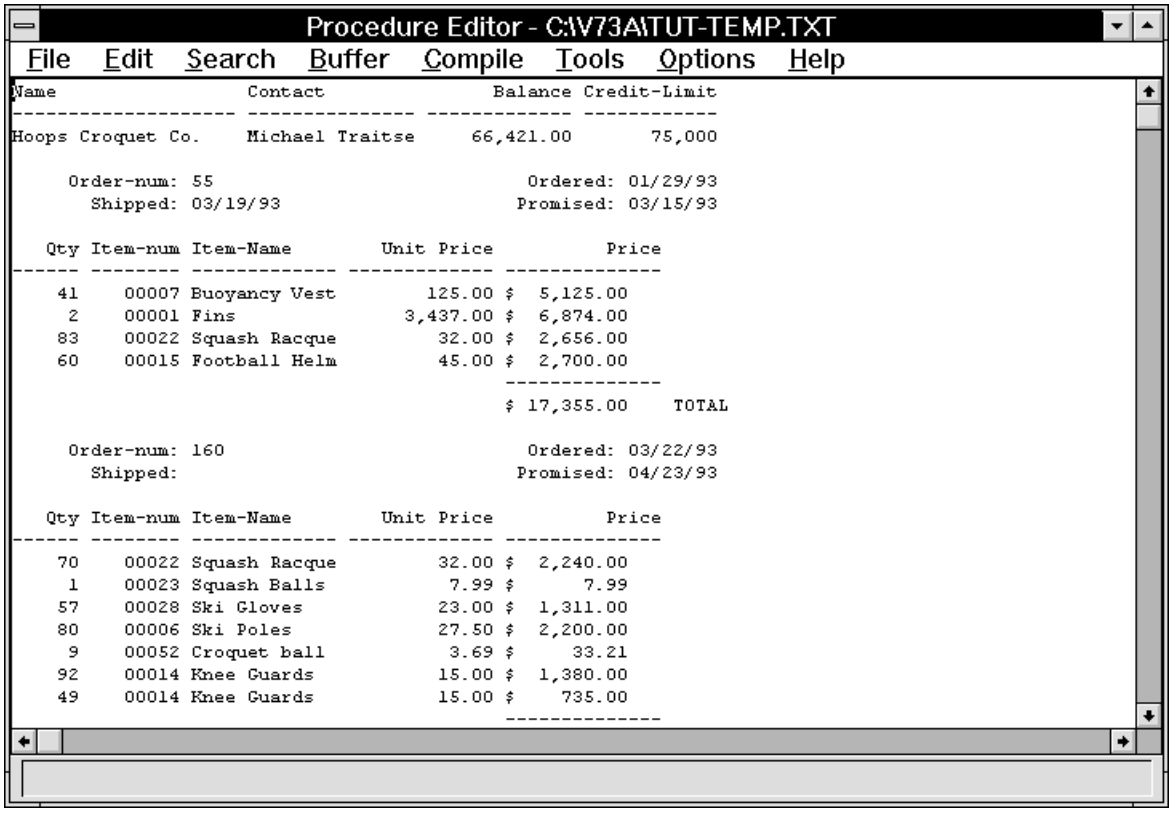
The i-10-09.p code example is simply the report from the last exercise, removed from the interface code.

i-10-09.p

```
/*1*/ OUTPUT TO "tut-temp.txt".
      FOR EACH Customer
        NO-LOCK WHERE Customer.Balance >= (Customer.CreditLimit * .85)
        WITH STREAM-IO:
        DISPLAY Customer.Name FORMAT "x(20)" Customer.Contact FORMAT "x(15)"
          Customer.Balance Customer.CreditLimit WITH NO-BOX STREAM-IO.
        FOR EACH Order NO-LOCK WHERE Order.CustNum = Customer.CustNum
          WITH STREAM-IO:
          DISPLAY Order.OrderNum Order.OrderDate Order.ShipDate
            Order.PromiseDate SKIP(1) WITH 2 COLUMNS STREAM-IO.

          FOR EACH OrderLine NO-LOCK
            WHERE OrderLine.OrderNum = Order.OrderNum WITH STREAM-IO:
            FIND Item NO-LOCK WHERE Item.ItemNum = OrderLine.ItemNum.
            DISPLAY OrderLine.Qty OrderLine.ItemNum
              Item.ItemName FORMAT "x(13)"
              Item.Price LABEL "Unit Price"
              Item.Price * Qty (TOTAL) LABEL "Price"
                FORMAT "$zzz,zz9.99 CR"
              WITH NO-BOX STREAM-IO.
          END. /* FOR EACH OrderLine */
        END. /* FOR EACH Order */
      END. /* FOR EACH Customer */
/*2*/ OUTPUT CLOSE.
```

You can see from the highlighted points that the `OUTPUT TO` and `OUTPUT CLOSE` statements control the stream. Also note that the `STREAM-IO` option must appear in each output frame phrase.

Try running `i-10-09.p` and then opening `tut-temp.txt` in the procedure editor. The following figure shows that the content of the file matches what the interface showed in the last exercise:



Directing output to a printer

To direct output to your default printer, use the OUTPUT TO statement with the PRINTER option. Remember to use the STREAM-IO option in your frame phrases. STREAM-IO removes all graphical characteristics of the widgets, leaving just the textual data.

You probably also want to use the PAGED option of the OUTPUT TO statement to create neat page breaks.

Try modifying i-10-09.p to send the report to a printer, if you have one available. Alternately, add the PAGED option to the OUTPUT TO statement, run the report, and open tut-temp.txt in the Procedure Editor.

Notice that the AVM redisplays the labels for the current frame after a page break.

To direct output to a printer other than the default printer, use the Printer (-o) startup parameter to designate a specified printer for the OpenEdge session. For more information, see the reference entry for the Printer (-o) startup parameter in the *Startup Command and Parameter Reference*.

Directing output to multiple destinations

ABL lets you specify multiple output destinations in a single procedure. You can use OUTPUT TO several times in a single procedure to direct the output to different destinations. You can also define named streams so you can output to several destinations.

First, you use the DEFINE STREAM statement to create the streams you need. This is a partial syntax for DEFINE STREAM:

Syntax

```
DEFINE STREAM stream-name
```

To output to the stream, you reference it in your output statement with the keyword STREAM, as shown in the following example:

```
/*1*/ DEFINE STREAM sToFile.

/*2*/ OUTPUT TO PRINTER.
/*3*/ OUTPUT STREAM sToFile TO "tut-temp.txt".

      FOR EACH Customer NO-LOCK:
/*4*/   DISPLAY Customer.Name Customer.Balance WITH STREAM-IO.
/*5*/   DISPLAY STREAM sToFile Customer.Name Customer.Balance WITH STREAM-IO.
      END.

/*6*/ OUTPUT CLOSE.
/*7*/ OUTPUT STREAM sToFile CLOSE.
```

The notes below describe how the code works:

- When the procedure starts, it has a default unnamed stream that outputs to the screen by default. This statement establishes a second stream that also outputs to the screen by default. Both streams are now available for the life of the procedure.
- This OUTPUT TO statement redirects the unnamed stream to the default printer.
- This OUTPUT TO statement redirects the named stream to output to a file.
- The first DISPLAY statement outputs to the unnamed stream only.

- The second DISPLAY statement outputs to the named stream only.
- This OUTPUT CLOSE statement redirects the default stream back to the screen.
- This OUTPUT CLOSE statement redirects the named stream back to the screen.

Designing frames for reports

What you've learned so far about reports covers the mechanics of generating, handling, and outputting data. This section covers some basics of report design. Report design, once again, is equivalent to frame design. Since a good portion of interface design also deals with frame design, you've already learned many techniques that can help make your reports look polished. The format phrase and frame phrase options that you relied on for designing interfaces also help with report design. For example, the AT, COLON, TO, SKIP, and SPACE options of the format phrase help you position widgets on the screen. They also let you position data in a report.

All the reports you have seen so far relied heavily on default ABL formatting. You'll want to design your key reports as carefully as you design your interfaces. Good report design means that you'll need the DEFINE FRAME statement to design your report containers. In addition to all the useful format and frame phrase options you learned earlier, there are some new techniques that you will learn about in this section, including:

- Using base fields as placeholders for variable expressions
- Using page headers
- Using page footers

Using base fields with the @ option

You cannot place an expression in a DEFINE FRAME statement because no values exist at compile time for ABL to format. Since you often need to put derived data in frames, you need a way to handle expressions. One solution is to create variables to hold the values. Another solution provides more flexibility: create variables with the characteristics of the expression results (data type, formatting, and label) as *base fields*. You can include the base fields in the frame definition and then replace the base field with any compatible expression result at runtime using the @ option of the DISPLAY statement.

Here is an example:

```
DEFINE Base1 AS CHARACTER NO-UNDO NO-LABEL.

DEFINE FRAME Frame1
  Name SKIP
  Base1 SKIP
  Phone SKIP
  WITH NO-BOX.
. . .
IF Contact EQ "" THEN
  DISPLAY Name "No Contact" @ Base1 Phone WITH FRAME Frame1.
ELSE
  DISPLAY Name Contact @ Base1 Phone WITH FRAME Frame1.
. . .
```

The previous example compiles a phone list of customer and contact names. If no contact is on file, "No Contact" appears in the report.

Using the HEADER of a frame for running page heads

A frame can contain three distinct sections: body, HEADER, and BACKGROUND. Until now, everything you placed in a frame became part of the frame body. Using the keywords HEADER and BACKGROUND, you can also define two additional sections for your frames. The following code fragment shows how a DEFINE FRAME statement with these sections would look:

```
DEFINE FRAME f-example
  body-item1         /* Can be field, variable, constant, image, or
  body-item2            or rectangle. */
  body-item3
  HEADER
    header-item1     /* Can be field, variable, constant, EXPRESSION,
    header-item2        image, or rectangle. */
    header-item3
  BACKGROUND
    background-item1 /* Can be field, variable, constant, EXPRESSION,
    background-item2    image, or rectangle. */
    background-item3
  WITH SIDE-LABELS.
```

Frame backgrounds are typically used for placing a logo (image) or other graphic device in the background of a display frame. However, you cannot print graphics from ABL. For more information, see *ABL Reference*. The rest of this section concentrates on the HEADER part of the frame.

The HEADER has a couple of special properties that allow you to implement running page heads and footers:

- A HEADER section can contain expressions.
- ABL re-evaluates expressions in a HEADER section each time the frame is redisplayed.
- ABL suppresses field and variable labels in a header frame. If you want labels, you supply text strings in the frame definition.

If a HEADER frame contains an expression, field, or variable, the frame definition must take place in the context where ABL can provide new values. In other words, for an iterating report procedure, move the DEFINE FRAME statement from the top of your procedure into the FOR EACH block. Think of a HEADER frame as an executable statement. Just like a DISPLAY statement inside a FOR EACH block, the HEADER section of the DEFINE FRAME statement executes on every iteration of the FOR EACH block.

Also, note that a frame does not have to have a body—it can consist of a header only. You can modularize your report design with three frames: one each for page header, body, and page footer. This approach lets you adopt standard headers and footers.

Assume that All Around Sports wants a standard page header on every page of its reports. This is the information they want to include in the page header:

```
Date: mm/dd/yy   Title    sales-rep  Page: xxx
```

The following code defines the first part of a procedure that implements the three-frame design.

```
/*1*/ OUTPUT TO "tut-temp.txt" PAGE-SIZE 25.
      FOR EACH Customer WHERE Customer.Balance >= 1400
        BREAK BY Customer.SalesRep WITH STREAM-IO:
        DEFINE FRAME f-hdr
/*2*/    HEADER
/*3*/      "Date:" TODAY "Customer Report" AT 25
/*4*/      Customer.SalesRep AT 55
           "Page" AT 65
/*5*/      PAGE-NUMBER FORMAT ">>9" SKIP(1)
/*6*/      WITH PAGE-TOP FRAME f-hdr STREAM-IO.
```

This part of the procedure contains the following language elements and points of interest:

- The PAGE-SIZE option of the OUTPUT statement sets the default size for a report page.
- The HEADER option tells the AVM to place the specified items in the header section at the top of the frame.
- The TODAY function returns the current system date, and constitutes an expression.
- The SalesRep initials come from the database and represent another part of the HEADER that the AVM must evaluate.
- The PAGE-NUMBER function tracks the current page number (expression).
- The PAGE-TOP FRAME f-hdr further defines what kind of HEADER the frame is. PAGE-TOP specifies where to place the frame and makes the frame a running page head.

Using the HEADER of a frame for running page footers

You can also use the HEADER section to create running page footers. The only difference is that you specify PAGE-BOTTOM in the frame phrase instead of PAGE-TOP. This is how All Around Sports wants their footers to appear:

```
Customer Report continued on next page
```

Once again, the report needs a DEFINE FRAME statement with the HEADER option as this code shows:

```
    DEFINE FRAME f-ftr
    HEADER
      "Customer Report"
      "continued on next page"
/*1*/ WITH FRAME f-ftr PAGE-BOTTOM CENTERED STREAM-IO.
```

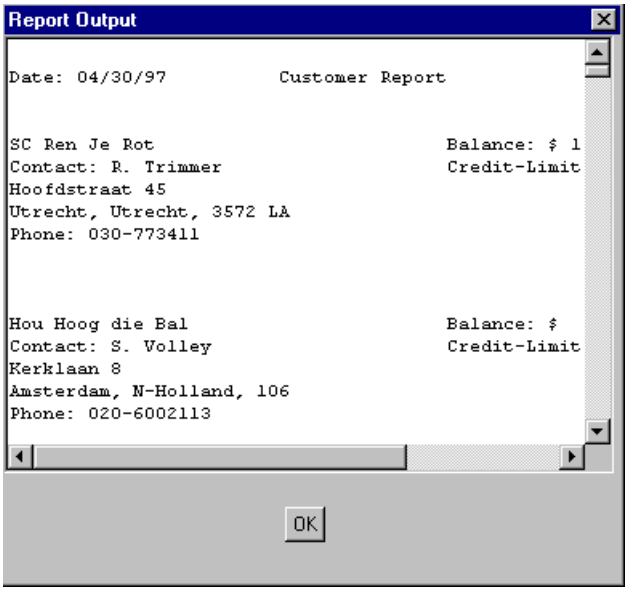
The PAGE-BOTTOM option tells the AVM to display the frame at the bottom of each page. PAGE-TOP and PAGE-BOTTOM frames are activated based on DISPLAY or VIEW statements. They are deactivated when the block to which the frames are scoped iterates or ends, which is why they have to be viewed in every iteration.

Programming example

The exercise in this section demonstrates the techniques available when using headers and footers in a report.

To see an example of using headers and footers in a report:

1. Open `i-10-10.p` and run it.
2. Click **Report**. The **Report Output** dialog box appears:



   Notice the header info at the top of each page. As you scroll through, you can see footers as well.

3. Click **OK**, then **Exit**, and then press **SPACEBAR** to return to the Procedure Editor.

The `i-10-10.p` code fragment shows the report generating code for this procedure.

i-10-10.p

```
      . . .
/*1*/  DEFINE FRAME f-body
          Name NO-LABEL
          Balance AT 40 FORMAT "$zzz,zz9.99 CR" SKIP
          Contact
          CreditLimit AT 40 FORMAT "$zzz,zz9.99 CR" SKIP
          Address NO-LABEL SKIP
          Holder NO-LABEL SKIP
          Phone SKIP(2)
          WITH SIDE-LABELS STREAM-IO.

/*2*/  OUTPUT TO "tut-temp.txt" PAGE-SIZE 25.
/*3*/  FOR EACH Customer
          WHERE Customer.Balance >= 1400 BREAK BY Customer.SalesRep:
/*4*/     DEFINE FRAME f-hdr
            HEADER
              "Date:" TODAY "Customer Report" AT 25
              SalesRep AT 55
              "Page" AT 65
              PAGE-NUMBER FORMAT ">>9" SKIP(1)
/*5*/       WITH PAGE-TOP FRAME f-hdr STREAM-IO.

/*6*/     DEFINE FRAME f-ftr
            HEADER
              "Customer Report"
              "continued next page"
/*7*/         WITH FRAME f-ftr PAGE-BOTTOM CENTERED STREAM-IO.

/*8*/     VIEW FRAME f-hdr.
          VIEW FRAME f-ftr.

          DISPLAY Name Balance Contact CreditLimit Address
/*9*/        (City + ", " + State + ", " + PostalCode) @ Holder Phone
             WITH FRAME f-body.

/*10*/    IF LAST-OF(SalesRep) THEN DO:
            HIDE FRAME f-ftr.
            PAGE.
          END.
        END. /* FOR EACH Customer */
        OUTPUT CLOSE.
        . . .
```

The following notes summarize the techniques shown in these topics:

- The body frame, which has no HEADER section, appears in its normal position, at the top of the file with other definitions.
- The PAGE-SIZE option sets the report page size.
- The use of the control break changes the report output from one report into a series of smaller reports-one for each sales rep.
- This HEADER frame comprises the running report head.
- PAGE-TOP places this frame at the top of the report page.
- This HEADER frame comprises the running page footer.
- PAGE-BOTTOM places the header frame at the bottom of the page.
- The VIEW statements force the AVM to evaluate the two HEADER frames on each iteration of the block.
- Here, the report creates an address string and uses the @ option to place the result at the Holder variable.

- The LAST-OF function is for checking for the end of a break group, allowing you to perform special tasks. In this case, the procedure suppresses the page footer because this break group report is complete. It also uses the PAGE statement to start a new page for the next break group report.

Using the PUT statement

The PUT statement is another useful ABL element for generating reports, especially when you want to customize certain parts of a report. The PUT statement has no default framing services, making it useful for writing data to a file or overriding default framing. Since PUT has no framing defaults, your procedures must contain explicit code for formatting your output.

The PUT statement outputs data one field at a time and uses the format of the field or variable. To include line breaks in the output, you must use the SKIP option. Additionally, the UNFORMATTED option of PUT displays all the data of the field or variable, regardless of format and without spaces between fields.

Why would you use PUT instead of DISPLAY? For every DISPLAY statement, ABL needs a frame. To execute a DISPLAY statement, the AVM builds a frame capable of handling the expected output, using default services and your explicit instructions. PUT on the other hand, simply outputs data one line at a time, with no default formatting. DISPLAY is most useful when you want automatic formatting. PUT is most useful when you want complete control over output.

This is a partial syntax for the PUT statement:

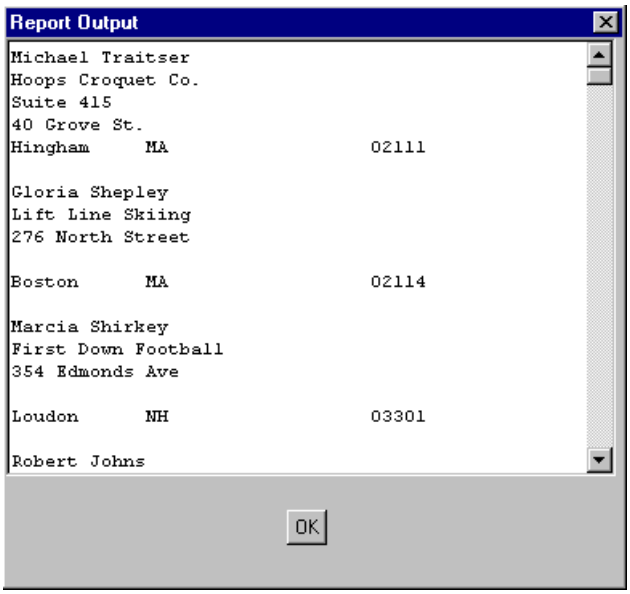
Syntax

```
PUT [ STREAM stream | STREAM-HANDLE handle ] [ UNFORMATTED ]
    [ expression [ FORMAT string ]
    [ AT expression | TO expression ]
      | SKIP [ ( expression ) ]
      | SPACE [ ( expression ) ]
    ] ...
```

One common task that the PUT statement can help with is mailing labels. Since mailing labels must conform to a compact physical layout and be uniform, using PUT is a good idea. Suppose that the All Around Sports accounting department wants to send notices to customers with large balances. They need a procedure that creates mailing labels for the notices.

To see an example that uses the PUT statement:

1. Open i-10-11.p and run it.
2. Click **Report**. The **Report Output** dialog box appears:



The editor shows the mailing list as it appears in a text file. Note that the output has the following problems:

- For simple addresses, there is a blank line between the street address line and the city, state, postal code line. Some addresses need this blank line for extra information.
- There is too much space between the state and postal code.

3. Click **OK**, then Exit, and then press **SPACEBAR** to return to the Procedure Editor.

Here is the code for this procedure:

i-10-11.p

```
    {i-10-in.i} /* Common Interface Setup Code */

    /**********  DEFINE TRIGGERS  **********/
    ON CHOOSE of b-rep DO:
/*1*/ OUTPUT TO "tut-temp.txt".
      FOR EACH Customer WHERE Customer.Balance >= 1400
        BY Customer.PostalCode:
/*2*/   PUT
          Customer.Contact SKIP
          Customer.Name SKIP
          Customer.Address SKIP
          Customer.Address2 SKIP
          Customer.City Customer.State Customer.PostalCode SKIP(1).
      END.
      OUTPUT CLOSE.

      ASSIGN Rep-Editor:READ-ONLY IN FRAME Dialog1 = TRUE
             Rep-Editor:SENSITIVE IN FRAME Dialog1 = TRUE
             FRAME Dialog1:TITLE                   = "Report Output"
             lStat = Rep-Editor:READ-FILE("tut-temp.txt") IN FRAME Dialog1.

      IF lStat THEN DO:
        ENABLE Rep-Editor b-ok WITH FRAME Dialog1.
        WAIT-FOR GO OF FRAME Dialog1.
        HIDE FRAME Dialog1.
      END.
    END.

    /**********  MAIN LOGIC  **********/
    ENABLE ALL WITH FRAME Frame1.
    WAIT-FOR CHOOSE OF b-exit.
```

This procedure contains the following points of interest:

1. The `OUTPUT TO` statement directs the output from this procedure to a text file named `tut-temp.txt`.
2. You can use the `PUT` statement in addition to the `DISPLAY` statement when sending data to a file or to a printer (any destination other than the screen).

To improve this procedure, you can:

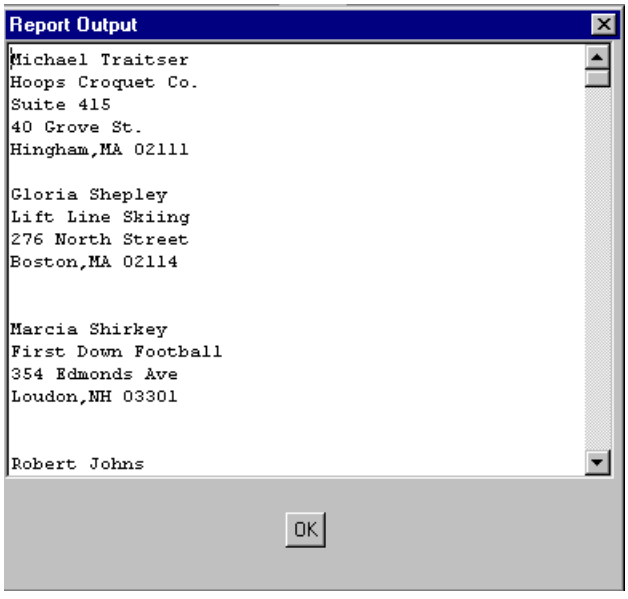
- Remove the blank line from simple addresses that do not use the extra space.
- Tighten up the spacing.

If you run `i-10-12.p`, you can see the modified version of this procedure.

i-10-12.p

```
    {i-10-in.i} /* Common Interface Setup Code */

    /**********  DEFINE TRIGGERS  **********/
    ON CHOOSE of b-rep DO:
      OUTPUT TO "tut-temp.txt".
      FOR EACH Customer WHERE Customer.Balance >= 1400
        BY Customer.PostalCode WITH STREAM-IO:
        PUT
          Customer.Contact SKIP
          Customer.Name SKIP
/*1*/     Customer.Address SKIP.

/*2*/   IF Customer.Address2 NE "" THEN
          PUT Customer.Address2 SKIP.

/*3*/   PUT Customer.City + "," + Customer.State + " " +
          STRING(Customer.PostalCode, "99999") FORMAT "x(23)" SKIP(1).

/*4*/   IF Customer.Address2 EQ "" THEN PUT SKIP(1).
      END. /* FOR EACH Customer */
      OUTPUT CLOSE.

      ASSIGN Rep-Editor:READ-ONLY IN FRAME Dialog1 = TRUE
             Rep-Editor:SENSITIVE IN FRAME Dialog1 = TRUE
             FRAME Dialog1:TITLE                   = "Report Output"
             lStat = Rep-Editor:READ-FILE("tut-temp.txt") IN FRAME Dialog1.

      IF lStat THEN DO:
        ENABLE Rep-Editor b-ok WITH FRAME Dialog1.
        WAIT-FOR GO OF FRAME Dialog1.
        HIDE FRAME Dialog1.
      END.
    END. /* ON CHOOSE OF b-rep */

    /**********  MAIN LOGIC  **********/
    ENABLE ALL WITH FRAME Frame1.
    WAIT-FOR CHOOSE OF b-exit.
```

The following is the output of `i-10-12.p`:



The following notes help explain the techniques used in the procedure:

1. The first PUT statement outputs and formats the part of the mailing label that is common to all labels.
2. The first IF statement determines whether the second address line has data. If it does, it outputs the data.
3. When you create a character expression, like the one in this PUT statement, the AVM removes trailing blanks from the fields. So this output tightens up the extra white space that showed up in the first mailing list example.
4. Finally, the second IF statement determines whether there is second address line data. If not, the PUT statement sends a blank line at the end of the address. This statement keeps the label data together and keeps the individual labels correctly spaced from each other.

Using PUT for printer control

When you send output to a printer, you may want to modify the way the printer generates that output. Many printers have a set of control sequences you can use to specify different print characteristics. You might, for example, want to change the number of printed characters per inch.

When you write a procedure that sends output to a printer, you can include printer control sequences within that procedure. Many control sequences involve special characters that can be represented by their octal (base 8) equivalent. To distinguish these octal codes, you precede the three octal digits by an escape character. the AVM then converts the octal number to a single character.

On UNIX, the escape character is a tilde (~) or a backslash (\).

The PUT statement with the CONTROL option allows you to specify a control sequence to send to the printer. This is a partial syntax for this version of the PUT statement:

Syntax

```
PUT [ STREAM stream-name | STREAM-HANDLE handle]
  CONTROL "ctrl-sequence" "ctrl-sequence" . . .
```

The control sequences you send to the printer have no effect on the current line, page counters, and positions maintained within ABL. Assume you want to print a report on your Brand X printer, using compressed-print mode.

The i-10-13.p procedure implements this task.

i-10-13.p

```
      /**********  DEFINE WIDGETS  **********/
/*1*/ DEFINE VARIABLE Start-Compress AS CHARACTER INITIAL "~033[3w".
/*2*/ DEFINE VARIABLE Stop-Compress  AS CHARACTER INITIAL "~032[3w".

      DEFINE BUTTON b-normal   LABEL "Print Normal Report".
      DEFINE BUTTON b-compress LABEL "Print Compressed Report".
      DEFINE BUTTON b-exit     LABEL "Exit".

      /**********  DEFINE FRAMES  **********/
      DEFINE FRAME Frame1
        SKIP(1)
        b-normal b-compress SKIP(1)
        b-exit
        WITH NO-BOX CENTERED THREE-D.

        /**********  DEFINE TRIGGERS  **********/
/*3*/ ON CHOOSE OF b-normal DO:
        OUTPUT TO PRINTER.
        FOR EACH Customer NO-LOCK
          WHERE Customer.Balance >= 1400 WITH STREAM-IO:
          DISPLAY Customer.Name Customer.Phone Customer.Balance
            Customer.SalesRep.
        END.
        OUTPUT CLOSE.
      END.

/*4*/ ON CHOOSE OF b-compress DO:
        OUTPUT TO PRINTER.
        PUT CONTROL Start-Compress.
        FOR EACH Customer NO-LOCK
          WHERE Customer.Balance >= 1400 WITH STREAM-IO:
          DISPLAY Customer.Name Customer.Phone Customer.Balance
            Customer.SalesRep.
        END.
        PUT CONTROL Stop-Compress.
        OUTPUT CLOSE.
      END.

      /**********  MAIN LOGIC  **********/
      ENABLE ALL WITH FRAME Frame1.
      WAIT-FOR CHOOSE OF b-exit.
```

These notes help to explain the code:

1. The start-compress variable contains the four-character sequence that puts the printer into compressed-print mode. These four characters are octal 033 (decimal 27) followed by left bracket ([), 3, and w.
2. The variable stop-compress takes the printer out of compressed print mode.
3. When the user clicks `b-normal`, the AVM runs the report in normal mode.
4. When the user clicks `b-compressed`, the AVM runs the report in compressed mode.

Note: This procedure works only if you have a printer connected to your system. Also, be aware that not all printers support compressed printing.

Control sequences are hardware-dependent, thus applications containing hard-coded printer control sequences are not easily portable to other environments. See *Manage ABL Applications* for more information about using the `PUT CONTROL` statement with control sequences.

Summary

The previous topics explained how to write ABL procedures that produce both simple and sophisticated reports.

Generating simple reports

You can use the DISPLAY statement to generate a simple report output to the screen from a single table or from multiple related tables. You can easily format a report to group records and include calculations with the CONTROL BREAKS option and aggregate functions.

Redirecting output

The STREAM-IO option allows you to redirect the output of a report-generating procedure to a printer or text file. This option removes the graphical components of your data. You can also redirect output to multiple destinations, and you can define multiple streams.

Generating reports of data represented by widgets

The widgets that deal with text-fill-in field and text widgets-pose no special problems when you generate reports. However, you must make certain adjustments for widgets that represent data graphically. There are two ways to handle these widgets in reports:

- Use the STREAM-IO option
- Use the PUT statement instead of DISPLAY; PUT suspends default frame-based formatting

Customizing reports

You can customize your reports by:

- Adding headers and footers
- Using advanced printing techniques, such as printer control sequences

III

External Program Interfaces

For details, see the following topics:

- Introduction to External Program Interfaces
- System Clipboard
- Providing Help for OpenEdge Applications
- Named Pipes
- Shared Library and DLL Support
- Windows Dynamic Data Exchange
- Using COM Objects in ABL
- ActiveX Automation Support
- ActiveX Control Support
- Sockets
- Host Language Call Interface

Host Language Call Interface

These topics describe the OpenEdge Host Language Call (HLC) Interface.

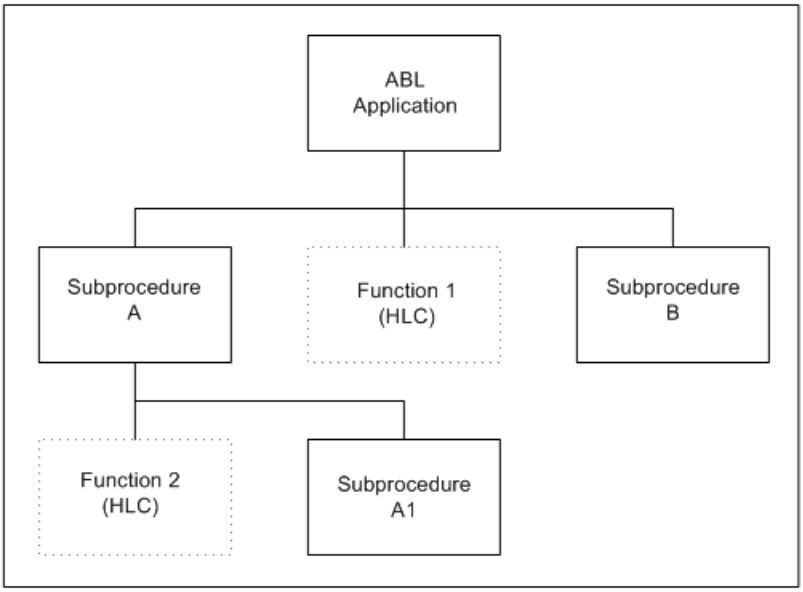
HLC and OpenEdge

The Host Language Call (HLC) Interface is an OpenEdge feature that allows you to write and call your own C language functions directly from an ABL (Advanced Business Language) procedure using the ABL CALL statement. Using HLC library functions, your C functions can read from or write to shared variables and shared buffer fields defined in the ABL context, interact with the ABL display, test for ABL interrupts, and access ABL-managed timer services. Ultimately, you can use HLC to access devices not supported directly in ABL (such as process sensors or ATM terminals), and exchange data between your ABL procedures and these devices.

HLC provides access to third-party application program interfaces from your ABL applications, similar to using shared libraries in the Windows environment (see Shared Library and DLL Support on page 483).

The following figure shows a top-down structure diagram for an OpenEdge application that calls your application functions using HLC.

Figure 62: OpenEdge application calling functions with HLC



Note: The HLC interface is supported both by the single-threaded ABL client and the PAS for OpenEdge multi-session agent.

Uses of HLC

The OpenEdge HLC Interface provides a way to call C language functions from within OpenEdge applications. You can use OpenEdge HLC to add portable custom features to OpenEdge. For example, you can link in C functions that do the following:

- Add support for trigonometric and other special numeric functions to OpenEdge
- Allow OpenEdge to access and control multimedia devices, such as video equipment
- Allow OpenEdge to directly access third-party proprietary data files, such as spreadsheet files

If you intend to port your extensions to other environments, use HLC. If you intend to build extensions for use with Windows only, use dynamic link libraries (DLL). If you intend to build extensions for use on Unix platforms, use Unix shared objects.

You can use shared libraries or DLLs with your OpenEdge Windows applications to call routines from an ABL procedure. An application links to these routines at run time rather than at build time, and shares the code with other applications. Any enhancement to a shared library or DLL immediately becomes available to your application without rebuilding.

For more information on using Unix shared libraries or Windows DLLs with OpenEdge, see Shared Library and DLL Support on page 483. See your Windows SDK documentation for details on building DLLs.

The HLC library functions provide an interface between C functions and ABL. From your C function, you can call HLC library functions that perform the following tasks:

- Read data from and write data to ABL shared or global variables
- Read data from and write data to ABL shared buffers
- Display OpenEdge-like messages
- Control interrupts
- Perform timer-service operations

For more information on using HLC library functions, see Use HLC library functions on page 391.

Requirements for using HLC

To be able to use HLC, you must be proficient with:

- The C language
- The C language compiler and linker on your computer

In addition, you need to make the following changes to your ABL and C code:

- Use the `CALL` statement in your ABL procedure when you need to make a C language function call. For information, see Use the CALL statement on page 377.
- Update `hlprodsp.c` to map your C function identifiers. For information, see Map routine identifiers using PRODSP() on page 378.
- Update `hlprodsp.c` to indicate whether or not your C source code is thread safe. You need to make this update only if the ABL procedure containing the `CALL` statement is deployed to a PAS for OpenEdge instance. For information, see Indicate whether the C source code is thread safe on page 381.
- Ensure the C source code is thread safe. For information on how to make code thread safe, see Guidelines for making the C source code thread safe on page 381.

Use the CALL statement

Use the ABL `CALL` statement to execute a C function from an ABL procedure.

This is the syntax for the `CALL` statement:

Syntax

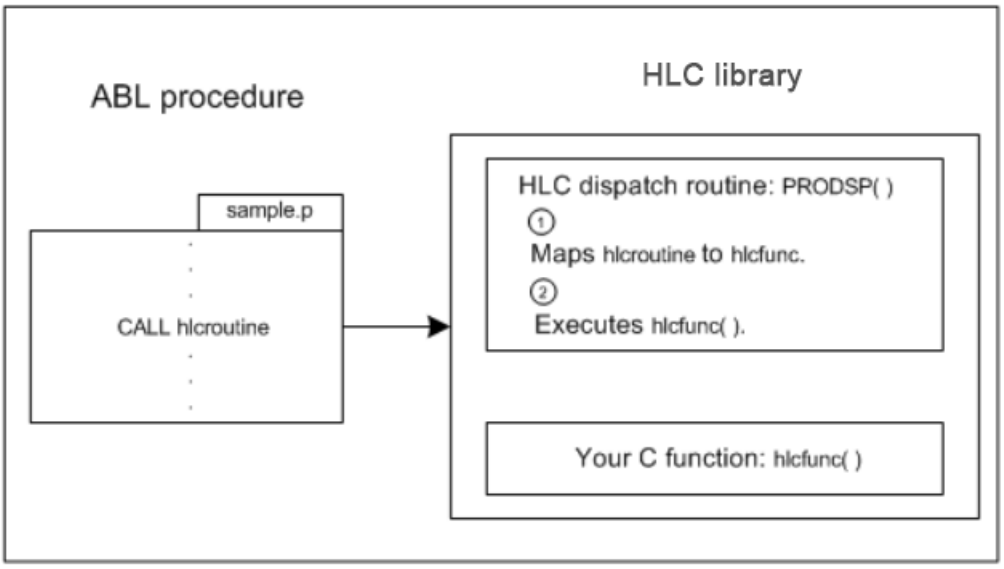
```
CALL routine-identifier [ argument ...]
```

The *routine-identifier* is the name the PRODSP( ) dispatch routine maps to an actual C function. The routine identifier is case sensitive; it must have the same letter case as its definition in the dispatch routine.

The *argument* is one or more arguments that you want to pass to the C function. Arguments passed to the C function must be read only. If you supply multiple arguments, separate them with spaces. If you separate them with other delimiters, such as commas, ABL passes the delimiters as arguments. ABL converts all arguments to C character strings before passing them to a C function; ABL passes them as an array of character pointers. Therefore, your C functions must expect null-terminated character strings and perform data conversions as necessary.

The following figure demonstrates how to run the CALL statement.

Figure 63: Running a CALL statement



The following CALL statement executes hlcroutine, the routine identifier for your C function:

```
CALL hlcroutine.
```

When you use a CALL statement to invoke a routine that updates a shared buffer, you must make sure that a transaction is active at the time of the call. For more information, see Use a library function that writes to a shared buffer on page 395.

The CALL statement transfers control to the HLC dispatch routine, PRODSP( ), passing it your routine identifier and any arguments. The example in the above figure passes the hlcroutine routine identifier with no arguments to PRODSP( ), which is located in hlprodsp.c.

You must modify the prototype hlprodsp.c file supplied with OpenEdge HLC to define your routine identifiers and C function names. Based on definitions you have set up in hlprodsp.c, the PRODSP( ) dispatch routine maps the routine identifier to your C function, and calls it. The example in the above figure maps hlcroutine to the function hlcfunc( ).

If the PRODSP( ) dispatch routine does not define the routine identifier or defines it with a different letter case, an error results.

Map routine identifiers using PRODSP()

You map routine identifiers to C functions in the PRODSP( ) dispatch routine. ABL provides a prototype PRODSP( ) in the C source file, hlprodsp.c.

Note: The $DLC/oebuild/hlc directory on and the %DLC%\oebuild\hlc directory in Windows contain a prototype hlprodsp.c file. Do not modify this file. To make changes, copy it to a working directory and modify the copy.

Because ABL calls the PRODSP() dispatch routine, it must have the following declaration:

```
PRODSP(pfunnam, argc, argv)

     char   *pfunnam,     /* Name of function to call */
     int    *argc,        /* CALL statement argument count */
     char   *argv[],       /* CALL statement argument list */
```

The hlprodsp.c file shows routine-identifier hlcroutine being mapped to the C function hlcfunc() in PRODSP():

hlprodsp.c

```
#define FUNCTEST(nam, rout) \
        if (strcmp(nam, pfunnam) == 0) \
            return rout(argc,argv);

/* PROGRAM: PRODSP
 *
 * This is the interface to all C routines that
 * ABL has associated 'CALL' statements to.
 */

long
PRODSP(pfunnam, argc, argv)

     char   *pfunnam;    /* Name of function to call */
     int     argc;       /* CALL statement argument count */
     char   *argv[];     /* CALL statement argument list */
{
    FUNCTEST("HLCROUTINE", hlcfunc);

    return 1; /* Non-zero return code causes OpenEdge error */
}             /* condition if CALLed routine not found. */
```

For each routine you add, you must include a call to the FUNCTEST macro. For example, to map two routine names, such as HLCROUTINE1 and HLCROUTINE2, to two corresponding C functions, such as hlcfunc1() and hlcfunc2(), you must include the following lines in PRODSP():

```
FUNCTEST("HLCROUTINE1", hlcfunc1);
FUNCTEST("HLCROUTINE2", hlcfunc2);
```

Note: The CALL statement and FUNCTEST declaration must use the same letter case for the routine identifier.

Syntax

```
FUNCTEST ( "routine-identifier" , function-name ) ;
```

This is the syntax for the FUNCTEST macro in hlprodsp.c:

The *routine-identifier* is the name referenced by a CALL statement that identifies your C function. Enter the routine identifier as a character string surrounded by quotes. Since FUNCTEST does not convert case for the routine identifier, the case is significant.

The *function-name* is the name of the C function that the *routine-identifier* references.

The routine identifier and the function name can have the same name, for example:

```
FUNCTEST("hlcfunc", hlcfunc);
```

When you compile hlprodsp.c, the C compiler translates the FUNCTEST macro references to C code, for example:

```
FUNCTEST("HLCROUTINE", hlcfunc);
```

Translates to:

```
if (strcmp("HLCROUTINE", pfunnam) == 0)
    return hlcfunc(argc,argv);
```

Therefore, when ABL invokes PRODSP( ) with the argument HLCROUTINE, it runs hlcfunc.

Indicate whether the C source code is thread safe

In addition to the dispatcher routine identifier mappings, the `hlprodsp.c` file contains the following functions which support HLC calls from the PAS for OpenEdge multi-session agent:

- `proHLC_IsThreadSafe`—A multi-session agent uses the value returned by this function at run time to decide whether calls to the HLC library (`prohlc.dll\libprohlc.so`) are serialized. If your code is thread safe, `proHLC_IsThreadSafe()` returns **1**. If your code is not thread safe, the function returns **0**, in which case the multi-session agent serializes calls to the HLC library. Set the value accordingly in the `hlprodsp.c` file.
- `proHLC_InitSession()`—This function is called after the HLC library is loaded and initialized. For a single-threaded ABL client, this function is called only once immediately after the library is loaded. For a multi-session agent, the function is called once for each ABL session immediately after the first HLC call is made from within an ABL session. If the custom C code is declared to be thread safe—that is, if `proHLC_IsThreadSafe()` is set to **1**, `proHLC_InitSession()` allocates a structure to store the non-constant global variables of the code and saves the structure in the current ABL session by calling `proSetGlobalStruct()`.

  The following is an example of using `proHLC_InitSession()`:

```
{
    void *ptr = promalloc(sizeof(struct myglobaldata));
    /* initialize myglobaldata structure with your non-constant */
    /* global data that you need access to for this ABL Session */
    proSetGlobalStruct(ptr); /* stash it so we can get it later */
}
```

  Note: Use `struct` *`myglobaldata`* to declare the non-constant global variables to which you need access in an ABL session where *`myglobaldata`* is a user-defined name of the structure.

- `proHLC_ExitSession()`—This function is called at the end of the ABL session to release the structure allocated in the `proHLC_InitSession()` function call.

  The following is an example of using `proHLC_ExitSession()`:

```
{
    struct myglobaldata *ptr = (void *)proGetGlobalStruct();
    /* First you should deallocate any allocated memory from myglobaldata */
    if (ptr)
        profree((void *)ptr); /* free up the structure itself */
    proSetGlobalStruct((void *) 0); /* so we don't dereference */
}
```

Guidelines for making the C source code thread safe

If the ABL procedure that has the ABL `CALL` statement is deployed to a PAS for OpenEdge instance, ensure the C source code is thread safe to take advantage of the multi-threaded functionality of the PAS for OpenEdge multi-session agent. Otherwise, C function calls are serialized.

Use the following guidelines to ensure your C source code is thread safe:

- Change calls to non-thread safe C run-time library functions in your C source code to thread safe implementations:
  1. Make a list of all the C run-time library function calls in the C source code.
  2. Classify the calls as thread safe and non-thread safe.

     Note: Interposing a function is the act of substituting an original function with a custom wrapper function.

  3. For non-thread safe function calls, ignore function calls that are interposed in the following Progress executables:
     - `_progres` for the ABL client
     - `_mproapsv` for the PAS for OpenEdge multi-session agent

     You are left with references to C run-time library functions that do not have thread safe implementations.

     For the complete list of C run-time library functions interposed in OpenEdge, see C run-time library functions interposed in the OpenEdge code on page 384.
  4. Change the remaining references to their thread safe versions.

- Reduce the usage of non-constant global variables in your source code:
  1. Make a list of all the non-constant global variables in your source code.
  2. Review the list.
  3. If there are global variables in the list which can be marked constant, then mark them so.

- Localize the remaining non-constant global variables:
  1. Move your global variables into a single structure.

     For example:

     ```
     struct myglobaldata { /* obviously, name this whatever you want*/
     int tls_foo; /* global called foo */
     struct datastruct *tls_pmydata; /* global called pmydata etc. */
     };
     ```

  2. Allocate and initialize the *myglobaldata* structure each time `proHLC_InitSession()` is called.
  3. Register *myglobaldata* with `proSetGlobalStruct()`.

     When `proHLC_ExitSession()` is called, clean up your global pointers and release *myglobaldata*.

     Whenever the C code needs to access a non-constant global variable, the code:

     1. Retrieves the allocated structure by calling `proGetGlobalStruct()`.
     2. Accesses the non-constant global variable in the retrieved structure.

  4. Define a thread variable for your *myglobaldata* structure. For example:

     On Unix

     ```
     __thread struct myglobaldata *pmygd; /* defined once */
     extern __thread struct myglobaldata *pmygd; /* everywhere you need access */
     ```

     On Windows

```
_declspec (thread) struct myglobaldata *pmygd; /* defined once */
_declspec (thread) __thread struct myglobaldata *pmygd; /* everywhere you need it
 */
```

5. Set the thread variable each time `PRODSP( )` is called. For example:

   ```
   pmygd = (struct myglobaldata *) proGetGlobalStruct(); /* retrieves */
   ```

6. Change global references so the code accesses them via the thread variable.

   For example, `foo = 10;` becomes `pmygd->tls_foo = 10;`. Since `pmygd` is set when `PRODSP( )` is called, it is available for as long as you need it.

7. If you have multiple global references, consider using a `#define` schema in your code. Include a `#define` schema wherever you need access to a global variable. For example:

   ```
   #define foo pmygd->tls_foo
   ```

- Avoid loading an extra DLL or shared object from the C source code. If you must load an extra DLL or shared object:
  - Confirm the DLL or shared object is thread safe.
  - Serialize the loading operation using synchronization mechanisms available in the C library.
  - Ensure the DLL or shared object is loaded only once and not by multiple threads.
- After you build the HLC library, perform the following checks:
  - Validate the global variables in the HLC library are thread safe (for example, using the `nm` command).
  - Validate the C run-time library function calls made by the HLC library are thread safe (for example, using the `nm` command). When you build your HLC library on AIX, include `csmtcrt.o` so that the library uses thread safe implementations of the C run-time library functions.

C run-time library functions interposed in the OpenEdge code

The following table lists the C run-time library functions in the OpenEdge code base made thread safe through interposition:

Table 71: C run-time library functions used in the OpenEdge code

CRT functions
ctime()
gmtime()
localtime()
gethostbyaddr()
gethostbyname()
getprotobyname()
getpwnam()
getpwuid()
getservbyname()
readdir()
strerror()
ttyname()
strtok()
rand()
random()
inet_ntoa()
getlogin()

The thread safe versions of these functions are by default interposed on all platforms except AIX. To ensure interposition on AIX, use the provided `csmtcrt.o` and the associated `csmtlibinit.o` file when building the HLC library.

The following table lists the thread safe versions of C run-time library functions in the csmtcrt.o file:

Table 72: Additional C run-time library functions and their thread safe implementations

Thread safe C run-time library function implementations	Non-thread safe C run-time library functions
psc_crypt()	crypt()
psc_setlocale()	setlocale()
localeconv_r()	localeconv()

If the C source code has references to crypt(), setlocale(), or localeconv(), change those references to their thread safe implementations.

HLC files and directories

The default HLC installation process creates the directories listed in the following table.

Table 73: HLC directories

Unix	Windows
$DLC/oebuild/hlc	%DLC%\oebuild\hlc
$DLC/oebuild/hlc/examples	%DLC%\oebuild\hlc\examples

The $DLC/oebuild/hlc directory contains the following files:

Table 74: HLC filenames

Filename	Description
hlc.h	HLC function library header file
hlprodsp.c	Prototype HLC dispatch routine file

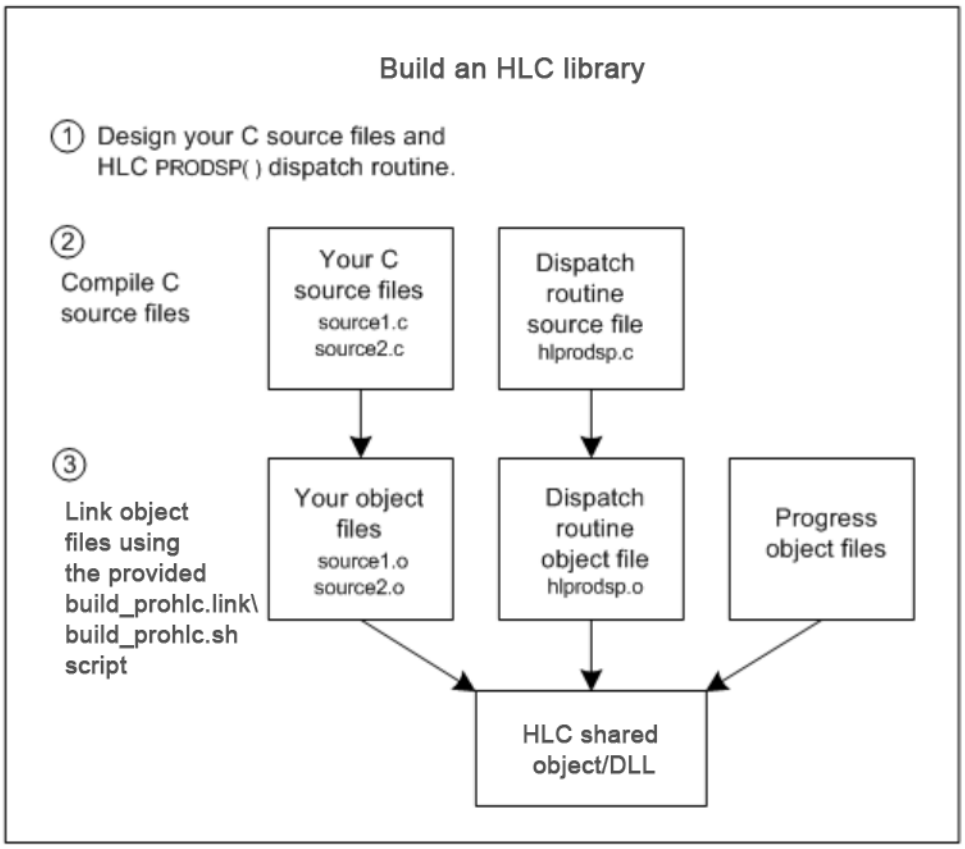
The $DLC/oebuild/hlc/examples directory contains HLC files for an oil tank demo application called Tank, which is described in Example HLC application.

Build an HLC library

After you write your C functions and modify the hlprodsp.c file to include the C function references, you build your HLC library (prohlc.dll for Windows and libprohlc.so for Unix).

The following diagram illustrates the steps required to build an HLC library:

Figure 64: Steps to build an HLC library



To build `prohlc.dll` or `libprohlc.so`, perform the following steps:

1. Set environment variables for the C complier on Windows or Unix.
2. Compile your local copy of `hlprodsp.c` and the source files that contain your C functions.
3. Edit your copy of the `build_prohlc.link\build_prohlc.sh` script to include the object files (`.obj` or `.o`) generated in Step 2.

   Note: Ensure that the Object files shipped with OpenEdge on page 387 are present in the `build_prohlc.link\build_prohlc.sh` script.

4. Run the `build_prohlc.link\build_prohlc.sh` script to build the HLC library (`prohlc.dll` or `libprohlc.so`).

   On Windows

   ```
   link @build_prohlc.link
   ```

   On Unix

   ```
   sh build_prohlc.sh
   ```

5. Set the new environment variable PROHLC to the full path of the newly built HLC library (`prohlc.dll` or `libprohlc.so`) including the library name.

On Windows

```
set PROHLC=<full_path_of_prohlc.dll>
```

On Unix

```
export PROHLC=<full_path_of_libprohlc.so>
```

Object files shipped with OpenEdge

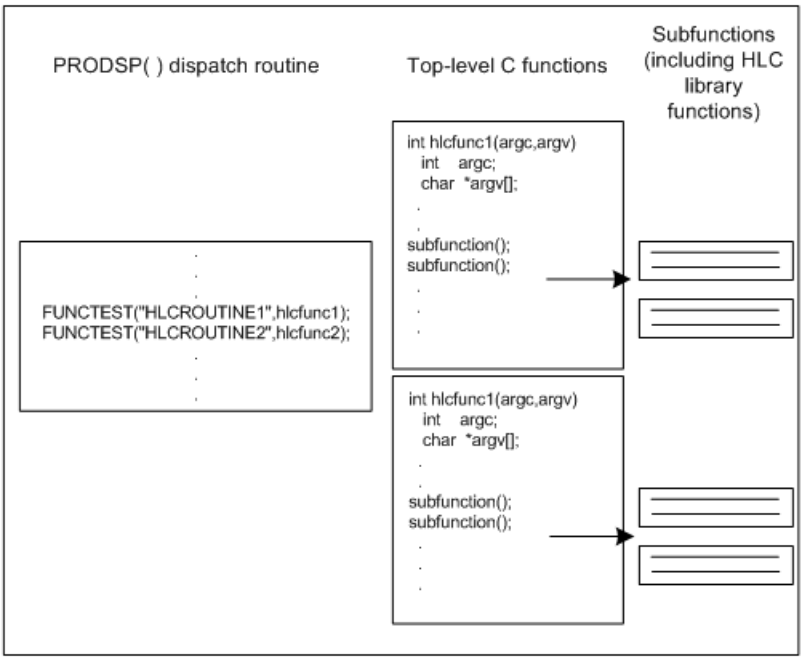
The following object files shipped with OpenEdge are required to build an HLC library:

- `prohlcdll.obj\prohlcdll.o`—In addition to the entry points for the HLC interface defined in `hlc.h`, this file holds the following entry points:
  - `void *promalloc(size)`—Returns the memory allocated to the current ABL session.
  - `void profree(void *ret)`—Frees the memory allocated to the current ABL session using `promalloc()`.
  - `void proSetGlobalStruct(void *ptr)`—Saves a pointer to a user-defined global data structure in the context of the current ABL session. The global structure is returned via a call to `proGetGlobalStruct*()`.
  - `void *proGetGlobalStruct()`—Returns the pointer to the memory saved by the call to `proSetGlobalStruct()`.
- `csmtcrt.o`—Contains Progress thread safe implementations of the C run-time library functions which are not thread safe. At run time, all calls from the HLC library to the C run-time library functions are routed to the interposed thread safe functions. This object file is required only on the AIX platform.
- `csmtlibinit.o`—Initializes the HLC library with `csmtcrt.o`. This object file is required only on the AIX platform.

Write C functions

The following figure shows the relationship between the PRODSP( ) dispatch routine that ABL supplies and your C functions.

Figure 65: Relationship between PRODSP() and your C functions



In general, your C functions can perform the following tasks:

- Call HLC library functions to read data from and write data to ABL shared buffers and variables. HLC library functions are C functions that ABL provides to access the OpenEdge application environment from a user-written C function.
- Perform operations using data from both OpenEdge and other sources. Other sources might include optical scanners, process monitor and control devices, spreadsheets, and other devices or applications not supported directly by OpenEdge.

Your C functions consist of functions called directly from PRODSP( ) (top-level functions), and functions called directly or indirectly from the top-level functions (subfunctions). All top-level functions must return control to ABL, regardless of the subfunctions called.

Top-level C function declaration

If you use the FUNCTEST macro provided in the prototype `hlprodsp.c` file, your top-level function must use the following syntax:

Syntax

```
long function-name ( arcg, argv )
  int   argc;
  char *argv[];
```

The length and content of the function name must follow the conventions that your C compiler and linker require. The *argc* and *argv* parameters follow the rules defined for *argc* and *argv* passed by the standard `main()` C function. If your top-level function must use a different interface, either write alternative dispatch code for it or rewrite FUNCTEST to meet your particular needs.

For more information on the FUNCTEST macro, see Map routine identifiers using PRODSP() on page 378.

Return error codes from a top-level function

There are two ways to return errors from a top-level function:

- Return from your top-level function with a non-zero value. ABL receives the non-zero value from `PRODSP()` and raises an error condition.
- Before returning from your top-level function, set an ABL shared variable to a specific value using an HLC library function.

If you use the first technique, the `PRODSP()` dispatch routine returns a long data type value to ABL. ABL uses this value as a return code to determine whether the HLC routine dispatched by `PRODSP()` was successful. If the return code is 0, ABL considers the routine successful and continues processing. If the return code is non-zero, ABL considers the routine unsuccessful and raises an error condition.

The `FUNCTEST` macro ensures that the value returned from your top-level function is passed on as the return code of `PRODSP()`. However, you must make sure that your top-level function returns an appropriate value for the return code. For example, when your top-level function runs successfully, make sure that the last statement executes the equivalent of the following:

```
return 0;
```

If you use the second method, you can set unique error codes that you can test from ABL and respond to each type of error as needed.

The first technique provides an efficient way to return to ABL when your HLC function encounters an unexpected error, such as an unsuccessful HLC library call. The second technique provides more fine-grained error processing, and allows your ABL procedure to handle conditions specific to your application.

Name C functions

You must name all external functions (entry points) so that your function names do not conflict with ABL entry point names.

On many systems, if you have an entry point in your code with the same name as an ABL entry point, the linker detects duplicate entry point symbols and displays an error message. However, on some systems the linker replaces one entry point with the other, and does not display a message.

No ABL entry point ends with _USR. Therefore, use the following syntax to guarantee that your function names do not conflict with ABL entry point names:

Syntax

```
function-name_USR ( ... )
```

The *function-name* is any name compatible with your development environment. For example, you might have a function called `calc_USR()`.

C function portability

C code is portable from one operating system to another when you can compile, link, and run it without change on either operating system. If you plan to port your HLC application, make your C functions as portable as possible. The following tips can help increase the portability of your C code:

- Avoid operating-system-specific calls. Proposed Unix-style portable operating system standards exist (POSIX, XOPEN), but their acceptance is limited. OpenEdge runs on operating systems that do not support these standards.
- If you must make operating-system-specific calls, hide them behind a standard application interface of your own design. If possible, confine the interface definition and all operating-system-specific code to a single source module, where you can modify it easily.
- Use the Unix lint utility if it is available on your system. The `lint` utility flags C language statements in your code that are potential sources of portability problems. See your system documentation for information on using lint or its equivalent.
- When you pass a value to a function, cast it to the data type the function expects. For example, the following function call casts the number 23 as a `long`:

```
sample_function((long)23);
```

- Do not use C functions when you can use standard ABL code.
- Do not write your own function if there is an HLC library function that provides the same functionality.

Avoid common HLC errors

To ensure compatibility between your HLC functions and ABL, consider the following when writing your functions:

- An HLC library function does not always return character and decimal data as null-terminated strings. For example, the *`pvar`* parameter of the prordc() function contains character data without null termination. To terminate such a string with the null character, you must add it yourself.
- When you check the value of a field or variable that HLC returns, check for the Unknown value (?). The field might not always contain a valid value.
- If you return from your top-level function without setting an appropriate return code value, the AVM might raise an error condition unexpectedly. See Return error codes from a top-level function on page 389 for more information.
- Use the `promsgd()` HLC library function to display messages directly to the terminal. Using the standard C function `printf()` might produce unexpected results when used to send raw data to a terminal.

Memory allocation

Use `malloc()` and `free()` to allocate and deallocate memory within a C function. Before returning to ABL, deallocate all memory that you allocate in the function.

Caution: If you allocate too much memory in an HLC function and leave it allocated after returning to ABL, your OpenEdge application might not have enough memory for its own needs. If OpenEdge runs out of memory at any point in the ABL application, it aborts and returns to the operating system with an error.

Data size

In order to run multiple instances of the OpenEdge executable in Windows, the static data size must not exceed 64K. The OpenEdge executable is a large model application that meets this requirement. However, if your HLC objects add excessive static data to the module, a linker error such as the following occurs:

```
DGROUP Greater than 64K.
```

To avoid this problem:

1. Dynamically allocate any large buffers or arrays.
2. Move your static text strings and constants into a resource file or into the code segment using _based pointers, as shown in the following example:

```
char _based(segname("_CODE")) mytext[] = "string of text";
```

See the Windows C Compiler documentation for details.

Use HLC library functions

The HLC library functions provide an interface between your C functions and ABL. From your C functions, you can call HLC library functions that perform the following tasks:

- Read data from and write data to ABL shared or global variables
- Read data from and write data to ABL shared buffers
- Display OpenEdge-like messages
- Control interrupts
- Perform timer-service operations

Access ABL data

This section describes how to use HLC library functions to access ABL data. For example, you can use the `prordbi()` function to read an integer field in a shared buffer. Use the following guidelines when accessing ABL data with HLC library functions:

- **Use correct data types for the parameters you pass to HLC library functions** — In general, C compilers do not verify whether the data types of the parameters you provide in a function call agree with the parameter data type definitions specified in the function declaration.

- **Use correct values for the parameters you pass to HLC library functions** — For example, HLC functions that access ABL shared buffers use the *fhandle* parameter. The *fhandle* parameter has the INTEGER data type. You must set *fhandle* correctly to read from or write to the correct location in the buffer.
- **Use pointer variables properly** — Many HLC library functions that access ABL use pointer variable parameters. Improper use of pointer variables can cause you to overwrite random locations in memory, with potentially hazardous results.

Caution: If you do not follow these guidelines, you can permanently damage your OpenEdge database.

The following example demonstrates these guidelines by defining shared buffer custbuf within an ABL procedure:

```
DEFINE NEW SHARED BUFFER custbuf FOR Customer.

FIND FIRST custbuf.
CALL subfunc1.
```

Later in your ABL code, execute a CALL statement that calls a C function. Within the C function, you read the custnum field. The custnum field is an integer field in the customer table, for which shared buffer hlcbuff is defined. For example:

```
#include "hlc.h"
subfunc1()
{
  int    ret, index, unknown;
  char    message[80];
  long    cnum;
  int    fhandle;

index = 0;
unknown = 0;

fhandle = profldix ("custbuf", "custnum");
ret = prordbi ("custbuf", fhandle, index, &cnum, &unknown);
if (ret||unknown)
{
  sprintf (message, "prordbi fatal ret = %d unknown %d", ret, unknown);
  promsgd (message);
}
sprintf(message, "customer.custnum was %ld", cnum);
promsgd(message);
return 0;
}
```

The prordbi() function reads an integer field contained in a shared buffer. To determine the HLC library function to use, see the function summary in HLC Library Function Reference on page 671.

This is the syntax for the prordbi() function:

Syntax

```
int prordbi ( pbufnam , fhandle , index , pvar , punknown )
  char *pbufname ;
  int   fhandle ;
  int   index ;
  long *pvar ;
  int  *punknown ;
```

The *pbufnam* parameter points to the name of the specified shared buffer. You supply the name from your OpenEdge application.

The *fhandle* input parameter is the field handle that `profldix()` returns for the specified field.

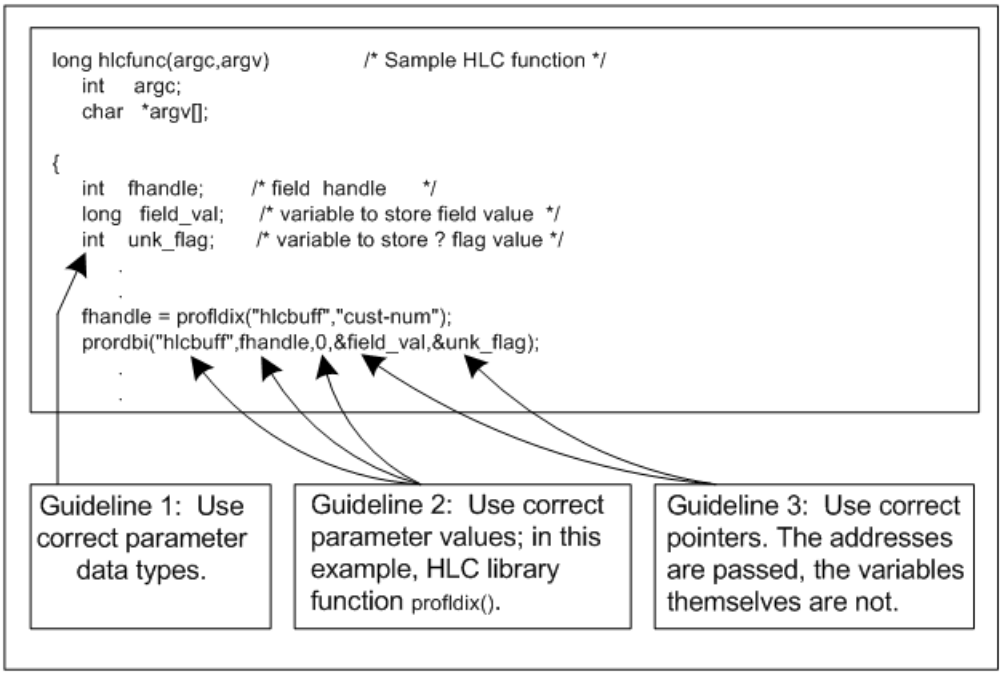
The *index* input parameter specifies an index value for an array field. If the field is a scalar, you must set the value of *index* to 0.

The *pvar* output parameter points to a long where `prordbi()` returns the value of the specified integer field.

The *punknown* output parameter points to an integer where `prordbi()` returns 1 if the field has the Unknown value (?), and returns 0 otherwise.

The following figure shows a call being made to `prordbi()` that illustrates HLC programming guidelines.

Figure 66: Example HLC library function call



Data type conversion

HLC data type conversion happens in two directions:

- From C language data types to ABL data types. This occurs when you write to an ABL shared variable or shared buffer.
- From ABL data types to C language data types. This occurs when you read from an ABL shared variable or shared buffer.

Convert C language data types to ABL data types

When you use an HLC library function to write data to an ABL shared buffer or shared variable, the HLC library function converts the C language data type passed to the function to the appropriate ABL data type.

For example, the `prowtbd()` HLC library function allows you to write to a date field in a shared buffer. Among the parameters you pass to `prowtbd()` are *`year`*, *`month`*, and *`day`*, all defined as integer in your C code. The `prowtbd()` function converts these separate C language integer values into a single ABL date value.

Convert ABL data types to C language data types

When you use an HLC library function to read data from an ABL shared buffer or shared variable, the HLC library function converts the ABL data type of the buffer or variable to a C language data type.

For example, the `prordbd()` HLC library function allows you to read from an ABL date field in a shared buffer. Among the parameters you pass to `prordbd()` are *`pyear`*, *`pmonth`*, and *`pday`*, all defined as pointers to integer in your C code. The `prordbd()` function converts the single date value within the buffer to separate C language integer values and inserts the values in the memory locations to which *`pyear`*, *`pmonth`*, and *`pday`* point.

See HLC Library Function Reference on page 671 for details on the parameters each HLC library function uses and the data types of these parameters.

Call an HLC library function

Any C source file that calls an HLC library function must include the `hlc.h` header file at the beginning of the file, as shown in the following example for Unix:

```
#include"$DLC/oebuild/hlc/hlc.h"
```

Note: This statement assumes you installed HLC in the default `$DLC/oebuild/hlc` directory for Unix.

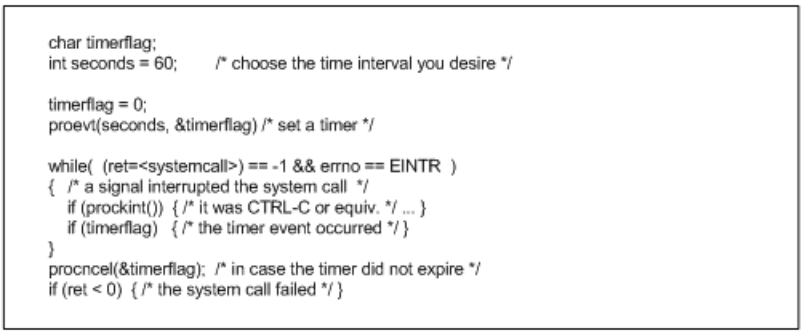
Timer services

Timer services provide functions that allow your application to wait for or respond to the completion of a specified time interval on the system clock.

Any HLC function that requires timer services should not use the `sleep()` or other operating system functions to access the system clock. Access the following HLC library functions for timer services on Unix: `prosleep()`, `proevt()`, `prowait()`, and `procncel()`. Access prosleep for timer services in Windows. The `proevt()`, `prowait()`, and `procncel()` timer services are available only on Unix. See HLC Library Function Reference on page 671 for information on these functions.

The timer-service functions provide all the tools you need to determine whether a timer is expired. The Unix-specific program fragment shows examples of timer-service functions `proevt()`, `prockint()`, and `procncel()`, as shown in the following figure.

Figure 67: Timer-service functions on Unix



```
char timerflag;
int seconds = 60;       /* choose the time interval you desire */

timerflag = 0;
proevt(seconds, &timerflag) /* set a timer */

while(  (ret=<systemcall>) == -1 && errno == EINTR  )
{  /* a signal interrupted the system call  */
   if (prockint())  { /* it was CTRL-C or equiv. */ ... }
   if (timerflag)   { /* the timer event occurred */ }
}
procncel(&timerflag);  /* in case the timer did not expire */
if (ret < 0)  { /* the system call failed */ }
```

User interrupt handling

The HLC function library contains an interrupt-handling function, `prockint()`, that allows you to check when a user presses the **STOP** key (**CONTROL-C** on Unix; **CONTROL-BREAK** in Windows).

Use `prockint()` in place of your own interrupt-handling function. Using your own interrupt-handling function during an HLC call could interfere with how ABL handles interrupts once you return from the call.

Use a library function that writes to a shared buffer

When you use a `CALL` statement to invoke a routine that writes to a shared buffer, you must make sure that a transaction is active at the time of the call. For example, place the `CALL` statement in a `DO TRANSACTION` block, as shown in the following code fragment:

```
DEFINE NEW SHARED VARIABLE errcode AS INTEGER NO-UNDO.

DEFINE NEW SHARED BUFFER newcust   FOR CUSTOMER.

FIND FIRST newcust.

/* 1 */
DO TRANSACTION:
  . . .
/* 2 */
  CALL HLCROUTINE1.
  IF errcode = 2 THEN DO:
    MESSAGE "Unable to read newcust".
    UNDO, RETRY.
  END.
  . . .
/* 3 */
END.
```

These notes explain the transaction block:

1. The transaction begins.
2. The `CALL` statement is defined.
3. The transaction ends.

The explicit DO TRANSACTION statement is not the only way to create a transaction in ABL. For example, you create an implicit transaction when you use the UPDATE statement within a FOR EACH block.

If your C function calls an HLC library function that writes to a shared buffer, (for example, prowtbc()), and no transaction is active, the library function returns a non-zero return code value that indicates an error has occurred and an ABL error condition results. Use an ON ERROR phrase within your ABL code to handle ABL error conditions.

Pass error codes back to ABL

You can use HLC library functions to pass a specific error code from your C function to ABL and test it for a specific value. To do this, define a shared variable within your ABL procedure to hold the error code value. Within your C function, set the shared variable with an HLC library function before returning to ABL. In the code fragment above, the shared variable errcode holds the error code value.

HLC applications on Unix systems

This section explains how to handle special features of Unix operating systems for HLC applications.

- Handle raw disk I/O on page 396
- Handle terminal I/O on page 396
- Handle abnormal exits on page 398

Handle raw disk I/O

On Unix systems that do not provide a synchronous write instruction, ABL uses raw disk I/O when writing to some of its key files. An HLC link script automatically modifies the permissions and ownership of the executable it creates. However, you might have to supply a root password when running the link script. If your OpenEdge executable does not have proper permissions, Unix displays a message similar to the following when you attempt to execute the new module:

```
Unable to use raw disk I/O
```

Handle terminal I/O

Character-mode systems support two basic modes of terminal I/O:

- **Raw** — Terminal I/O without any operating system processing. Set your terminal to raw mode using the [-]raw option of the stty command.
- **Cooked** — Terminal I/O that the operating system processes. Set your terminal to cooked mode using either the [-]raw or [-]cooked option of the stty command.

In raw mode, the system reads input characters immediately and passes them to an application without any interpretation, and without sending them to the display as they are entered. Also, the system does no preprocessing or postprocessing of output characters.

In cooked mode, the system interprets input characters according to the following terminal input functions that the Unix system defines: ERASE, KILL, INTR, QUIT, SWITCH, and EOT. Also, UNIX systems provide terminal-specific postprocessing, such as defining special characters and character mapping, echoing input to the display, etc.

Use the proscopn(), prosccls(), and promsgd() functions

In character mode, OpenEdge typically uses raw terminal I/O, but supplies HLC library functions for switching between raw and cooked mode. Ideally, leave the terminal set to raw mode and use the `promsgd()` function to display information to the user. This provides a uniform appearance for messages.

If necessary, you can use cooked mode for handling the screen display in character mode. In this case, use the `proscopn()` function to enable cooked mode. Before returning to OpenEdge, re-enable raw mode with a call to `prosccls()`.

In character mode, the `promsgd()` function displays up to two messages before displaying a "Press space bar to continue" status message. In raw mode, the user can press **SPACEBAR** as prompted. In cooked mode, the user must press **RETURN** after `promsgd()` displays the status message. To use `promsgd()` in cooked mode, use the ABL PAUSE statement to change either the status message or the way the user interacts with the display.

For example, to change the status message to tell the user to press **RETURN**, place the following PAUSE statement anywhere before the first HLC call that invokes `promsgd()`:

```
PAUSE BEFORE-HIDE MESSAGE "Press [RETURN] to continue"
...
CALL HLC-MESSAGE-ROUTINE.
```

To cause the display to change two seconds after displaying a message without user input, use the following statement:

```
PAUSE 2 BEFORE-HIDE NO-MESSAGE.
...
CALL HLC-MESSAGE-ROUTINE.
```

In this case, you must specify the interval to pause. See the PAUSE Statement reference entry in *ABL Reference* for more information.

Run the HLC demo application to see examples that include a number of promsgd() calls illustrating these techniques.

In graphical interfaces, `promsgd()` displays messages in an alert box. Raw and cooked terminal I/O, and the `proscopn()`, `proclear()`, and `prosccls()` functions apply only to OpenEdge running in character interfaces.

Handle abnormal exits

In character mode, if you abort to the operating system during application testing, or otherwise must abort from your production version, make sure your HLC application does not terminate with raw terminal I/O. Set the terminal to cooked terminal I/O before exiting. This prepares the terminal for Unix operation, as shown in the following example:

```
/*
 * Example of a fatal error exit for an HLC application.
 *
 * Note how the terminal is returned to cooked mode before
 * exiting. Note also that exiting in this fashion can be
 * hazardous to your database. (This function should only
 * occur in program development phase for convenience in the
 * debugging process.)
 */

void hl_fexit (code)
{
  proscopn(); /* restore terminal to cooked mode */
  exit (1);
}
```

Caution: An abnormal exit function might be useful during testing, but you should never include it in a production executable. Since it bypasses OpenEdge shutdown processing, this function can cause irrecoverable database damage.

OpenEdge automatically restores cooked-mode terminal I/O before exiting. If your application aborts in raw mode, the user can enter commands but might not see the result.

To reset a character-mode terminal in raw mode at a Unix shell prompt:

1. Press **CONTROL-J** several times. Shell prompts appear on the display.
2. If the stty command on your version of Unix has a `sane` option, enter the following command, followed by **CONTROL-J**:

   ```
   stty sane
   ```

   If your stty command does not have a `sane` option, enter the following command, followed by **CONTROL-J**:

   ```
   stty echo icanon icrnl opost
   ```

Sample HLC application

This topic explains how to use the Tank sample application. This application is shipped with OpenEdge that you can use to verify that your HLC environment is set up correctly. The C code and sample output for this sample application are provided in the following directories:

On Unix

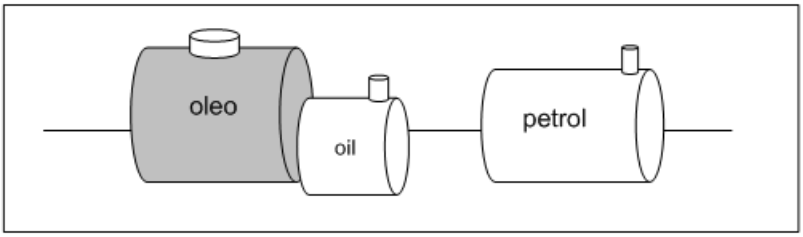
```
$DLC/oebuild/hlc/examples
```

On Windows

```
%DLC%\oebuild\hlc\examples
```

The Tank sample application uses OpenEdge to keep track of oil storage tanks. The ABL procedure calls a C program, AVCALC, to calculate the available capacity for a given oil storage tank. The following figure shows that the tanks are cylindrical, with their axes parallel to the level ground.

Figure 68: Tank positioning and orientation



To calculate the available capacity (empty portion) of the tank, you need to know the diameter, length, and current level of oil for the tank.

$$\text{length} \cdot r \cdot \left[\frac{p}{2} + \sqrt{1 - \left(1 - \frac{\text{level}}{r}\right)^2} \left(1 - \frac{\text{level}}{r}\right) + \sin^{-1}\left(1 - \frac{\text{level}}{r}\right) \right]$$

Where

r

Radius of the tank

length

Length of the tank

level

Level of oil in the tank

For this example, assume there is a tank table for this sample application that contains the following decimal fields:

radius

Radius of tank

tlength

Length of tank

depth

Level of oil—must be between 0 and (2 . radius)

tavail

Available volume in tank

In addition to these fields, the `tank-id` character field is used as the primary index.

The following example shows the Data Dictionary report for the Tank sample application.

Data Dictionary report for the Tank sample application

```
            Default field order: yes
09/10/93                  Data Dictionary Report                 Page   1
Database: tank
                                    tank File
                                    =========
                    (Flat file containing oil tank information)
Delete Validation
    Criterion:
      Message:
    Field        Type  Ext Dec Format                          Init
    ------------ ----- --- --- ------------------------------ ----------
*   tank-id      char          x(8)
    radius       dec         2 ->>,>>9.99                     0
    tlength      dec         2 ->>,>>9.99                     0
    depth        dec         2 ->>,>>9.99                     0
    tavail       dec         2 ->>,>>9.99

Index Name   Unique Field Name   Seq Ascending abbreviate
  ------------ ------ ------------ --- --------- ----------
# tank         yes    tank-id        1 yes       yes

Field Validation Criteria, Validation Messages
------------------------------------------------------------------------
depth        :  depth le 2 * (input radius)
                Depth cannot exceed diameter of tank

Help Messages
------------------------------------------------------------------------
tank-id      :  Tank identification number
radius       :  Radius of oil tank
tlength      :  Length of oil Tank
depth        :  Depth of oil in tank.
tavail       :  Available Volume In Tank


                        Data Dictionary Report Legend
             * - Indicates that a field participates in an index
             # - Indicates the primary index for a database file
             M - Indicates that a field is mandatory
```

A C function calculates the `tavail` field from the other three decimal fields (`radius`, `tlength`, `depth`). The following example shows the ABL procedure that invokes the C function by calling the HLC routine, AVCALC.

ABL procedure calling HLC routine AVCALC

```
/* calculate available volume for each tank */
DEFINE NEW SHARED BUFFER tankbuf FOR tank.

FOR EACH tankbuf:
  DISPLAY
    radius SPACE(3) tlength SPACE(3) depth SPACE(3) tavail
    WITH CENTERED TITLE "Tank Table".
END.
PAUSE 0.
VIEW FRAME tank-before.
HIDE ALL.

FOR EACH tankbuf:
  DO TRANSACTION:
    CALL AVCALC.
  END.

  DISPLAY
    radius SPACE(3) tlength SPACE(3) depth SPACE(3) tavail
    WITH CENTERED TITLE "After Calculation".
END.
```

Make a copy of the HLC dispatch routine, `hlprodsp.c`, and name it `tankdsp.c`. Modify the routine so an entry appears for AVCALC which calls the C subroutine `hlvcalc`.

The `hlprodsp.c` example shows the modifications to the `tankdsp.c` routine.

hlprodsp.c

```
#define FUNCTEST(nam, rout) \
         if (strcmp(nam, pfunnam) == 0) \
             return rout(argc,argv);

/* PROGRAM: PRODSP
 *
 * This is the interface to all C routines that
 * ABL has associated 'call' statements to.
 */
long
PRODSP(pfunnam, argc, argv)
    char   *pfunnam;
    /* Name of function to call */
    int     argc;
    char   *argv[];{
    /* Interface to 'tank' example */
    FUNCTEST ("AVCALC", hlvcalc);

    return 1;
}
```

The following procedure shows the code for the demo program `hlvcalc.c`. The program extracts the radius, length, and level fields from the shared buffer tank, calculates the available volume, and updates the `tavail` field in the shared buffer tank with the number calculated.

hlvcalc.c

```
 *   Obtains the height, radius and oil level for the tank
 *   from the shared buffer "tankbuf".
 *
 *   Calculates the remaining available volume
 *
 *   Update the avail field in the shared buffer "tankbuf" with the
 *   number calculated.
 */
#define BUFLEN 100
#include <math.h>
#include "hlc.h"
/*NOTE: M_PI_2 is pi/2 constant which may be defined in math.h */
#ifndef M_PI_2
#define M_PI_2 1.570796327
#endif

extern double asin();

char *fieldnm[] = { "tlength", "depth", "radius"};
char message[80];
int
hlvcalc()
{
   char     buffer[BUFLEN];
   int      unknown = 0, index = 0, varlen = BUFLEN, actlen;
   int      ret;
   double   length, depth, radius, avail;
   int      i;
   int      fldpos;
   double   temp1, temp2; /* used to simplify calculation */
   /* first, obtain the length, depth and radius from */
   /* the shared buffer "tankbuf". */
   for (i = 0; i < 3; ++i)
   {
      fldpos = profldix("tankbuf", fieldnm[i]);
      if (fldpos < 0)
      {
        sprintf(message, "profldix failed on %s for field %s",
                "tankbuf", fieldnm[i]);
        promsgd(message);
        return 1;
      }

ret = prordbn("tankbuf", fldpos, index,
                     buffer, &unknown, varlen, &actlen);
      if (ret)
      {
          sprintf(message, "prordbn failed accessing %s . %s",
                   "tankbuf", fieldnm[i]);
          promsgd(message);
          return 1;
      }
      /* if one of the fields is unknown, set avail field */
      /* to the unknown value */
      if (unknown)
      {
          fldpos = profldix("tankbuf", "tavail");
          if (fldpos < 0)
          {
              sprintf(message, "profldix failed on %s for field %s",
                      "tankbuf", "tavail");
              promsgd(message); return 1;
          }
        ret = prowtbn("tankbuf", fldpos, index, buffer, unknown);
          if (ret)
          {
              sprintf(message, "prowtbn failed, ret = %d", ret);
              promsgd(message);
```

```
            return 1;
         }
         return 0;

      /* convert the character string obtained from */
      /* ABL into a decimal number */
      buffer[actlen] = '\0';

      switch (i)
      {
         case 0:
                  length = atof(buffer); break;
         case 1:
                  depth = atof(buffer); break;
         case 2:
                  radius = atof(buffer); break;

         default:
                  break;
      }
/* Now, calculate the available volume */
/* NOTE: M_PI_2 is pi/2 constant defined in math.h */
#ifndef M_PI_2
#define M_PI_2 1.57
#endif
temp1 = 1.0 - depth/radius;
temp2 = temp1 * sqrt(1.0 - temp1 * temp1) + asin(temp1);
avail = length * radius * radius * (temp2 + M_PI_2);

/* Now, put this value in the tavail field in the */
/* "tankbuf" shared buffer */
/* get the double into character format */
sprintf(buffer, "%.2f", avail);
fldpos = profldix("tankbuf", "tavail");
if (fldpos < 0)
{
      sprintf(message, "profldix failed on %s for field %s",
                                    "tankbuf", fieldnm[i]);
      promsgd(message);
      return 1;
}
ret = prowtbn("tankbuf", fldpos, index, buffer, unknown);
if (ret)
      {
          sprintf(message, "prowtbn failed, ret = %d", ret);
          promsgd(message);
          return 1;
      }
return 0;
```

Run the Tank sample application on Windows

Run the Tank sample application shipped with OpenEdge to verify HLC is installed correctly on Windows.

Perform the following steps to run the Tank sample application:

1. Log on as a system administrator.
2. Copy the `%DLC%\oebuild\hlc\examples`, `%DLC%\oebuild\make`, and `%DLC%\oebuild\obj` directories to your working directory.

3. Set the environment variables for the C compiler. For example, run the following command in a Proenv window to set environment variables for Microsoft Visual Studio 2017:

   ```
   "C:\Program Files (x86)\Microsoft Visual
   Studio\2017\Professional\VC\Auxiliary\Build\vcvars64.bat"
   ```

4. Change your current directory in the Proenv window to <*working_directory*>\examples.
5. Compile the .c files in <*working_directory*>\examples by executing the following command:

   ```
   cl -c *.c
   ```

   As a result, a .obj file is generated for each .c file in the directory.

6. Edit the build_prohlc.link file in <*working_directory*>\make to include the newly generated .obj files. For example, change "./hlprodsp.obj" to "../examples/*.obj".
7. Change the current directory in the Proenv window to %DLC%\oebuild\make.
8. Generate the prohlc.dll file by executing the following command:

   ```
   link @build_prohlc.link
   ```

   prohlc.dll is generated in the current working directory.

9. Set the new environment variable PROHLC to the full path of the newly built HLC library (<*working_directory*>\make\prohlc.dll) by executing the following command:

   ```
   set PROHLC=<working_directory>\make\prohlc.dll
   ```

10. Change the working directory to <*working_directory*>\examples.
11. Create an empty OpenEdge database hlcdemo by executing the following command:

    ```
    prodb hlcdemo empty
    ```

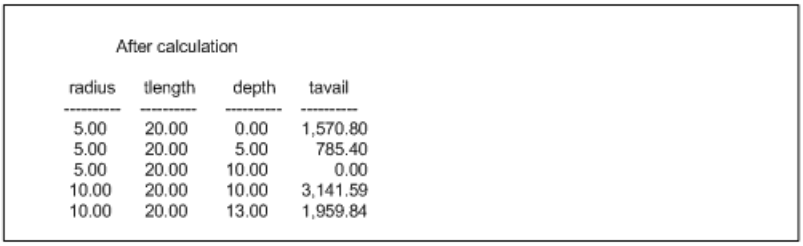
12. Load the data required for the sample application by executing the following command:

    ```
    prowin -db hlcdemo -1 -p hldemo.p
    ```

13. Finally, test the sample application by executing the following command:

    ```
    prowin -db hlcdemo -1 -p hltank.p
    ```

If you see the following output, it means that the sample application is working properly:



After calculation

radius	tlength	depth	tavail
5.00	20.00	0.00	1,570.80
5.00	20.00	5.00	785.40
5.00	20.00	10.00	0.00
10.00	20.00	10.00	3,141.59
10.00	20.00	13.00	1,959.84

Run the Tank sample application on Unix

To verify HLC is installed correctly on Unix, run the Tank sample application.

To run the sample application on Unix:

1. Log on as a system administrator.
2. Set the environment variables for the C compiler.
3. Copy the example HLC files to your working directory:

```
cp $DLC/oebuild/hlc/examples/* .
```

Note: Before you develop your own applications, move c and `build_prohlc.sh` to a directory that is in your path and move `hlc.h` to your include file directory or your current working directory.

4. Run the `testhlc` script in the command line.

The `testhlc` script creates an empty database, compiles the C source files to create object files in your current directory, links your test HLC module, and runs a small sample application.

After you run `hltank.p`, verify the results you obtain are correct by comparing them with the following results, which list the input values for radius, tlength, and depth, and the correct output for tavail:

```
After calculation

radius     tlength    depth      tavail
---------- ---------- ---------- ---------
  5.00       20.00       0.00   1,570.80
  5.00       20.00       5.00     785.40
  5.00       20.00      10.00       0.00
 10.00       20.00      10.00   3,141.59
 10.00       20.00      13.00   1,959.84
```

The following code example is the `testhlc` script on Unix:

```
#!/bin/sh
#  testhlc -- single-user test of HLC

#BLOCK 1
DLC=${DLC-/usr/dlc};export DLC
DLCDB=${DLCDB-$DLC};export DLCDB
OEBUILD=${OEBUILD-$DLC/oebuild};export OEBUILD

#BLOCK 2
SOURCEDB=$DLCDB/empty
DBNAME=hlcdemo
rm -fr $DBNAME.*
$DLC/bin/prodb $DBNAME $SOURCEDB
if [ $? -ne 0 ]
then
 echo "Failed to create database"
 exit 1
fi

#BLOCK 3
echo Now compiling *.c ...
c *.c
mv hlprodsp.o $OEBUILD/obj/

#BLOCK 4
echo Now making the test hlc shared object ...
HLC_OBJS="`ls -1 *.o`"
export HLC_OBJS
. $OEBUILD/make/build_prohlc.sh

export PROHLC=libprohlc.so

#BLOCK 5
rx $DBNAME -1 -p hldemo.p

if [ -r hlcout ]
then

 if cmp -s hlcout savehlc
 then
   echo "** HLC Installation Test has Completed Successfully."
 else
   diff hlcout savehlc >testhlc.diff
   echo "** HLC Installation Test reports conflicts."
   echo "   Please look at files:"
   echo "      'hlcout.dif', 'hlcout' & 'savehlc'"
 fi
fi
```

The following notes explain the blocks in the Unix `testhlc` script:

1. Sets up your Unix environment.
2. Creates and starts an empty database.
3. Compiles all the `.c` files necessary to run the sample application.
4. Links and loads the object files.
5. Starts an OpenEdge session and runs the `hldemo.p` procedure.

Source code listings

This section shows some of the source files provided in the `$DLC/oebuild/hlc/examples` directory.

This `loaddb.p` procedure automatically loads the data definitions for the sample application.

loaddb.p

```
hide all.
display skip(2)
        " ********* LOADING the database ********* " skip(1)
        " ********* please stand by............."
        with title "WELCOME TO HLC DEMO PROGRAM"
                 row 5 centered frame hdr.

output to hldemout.
run prodict/load_df.p ("customer.df").
run prodict/load_df.p ("agedar.df").
run prodict/load_df.p ("monthly.df").
run prodict/load_df.p ("tank.df").
/* create signal file to indicate success */
if opsys = "unix" then do:
   unix cat >hlcsigfile & .
end.
hide all.
```

The `hlprodsp.c` file is the ABL prototype in which you define the application's HLC routine identifiers and the corresponding C function names.

hlprodsp.c

```
#include "hlc.h"
/*   File:   hlprodsp.c              */

#define FUNCTEST(nam, rout)  \
        if (strcmp(nam, pfunnam) == 0) \
     return rout(argc,argv);

/* PROGRAM: PRODSP
 *
 *   This is the interface to all C routines that
 *   PROGRESS has associated 'call' statements to.
 */

long
PRODSP(pfunnam, argc, argv)

     char   *pfunnam;    /* Name of function to call */
     int     argc;
     char   *argv[];

{
/* Interface to installation test */
    FUNCTEST( "SUBVRD", subvrd)
    FUNCTEST( "SUBVWT", subvwt)
    FUNCTEST( "SUBFRD", subfrd)
    FUNCTEST( "SUBFWT", subfwt)
    FUNCTEST( "SUBFIX", subfix)
    FUNCTEST( "SUBVIX", subvix)
    FUNCTEST( "SUBARG", subarg)
    FUNCTEST( "SUBCLR", subclr)
    FUNCTEST( "SUBCLS", subcls)
    FUNCTEST( "SUBINT", subint)
/* Interface to 'screen' examples */
    FUNCTEST ("EXAMPLE1", example1);
    FUNCTEST ("EXAMPLE2", example2);
/* Interface to 'tank' example  */
    FUNCTEST ("AVCALC", hlvcalc);
    return 1;
}

int proHLC_IsThreadSafe()
{
    return 0; /* return 1 if you've confirmed that your code is thread safe */
}

struct myglobaldata
{
 int a;
/*
 * Put your non-constant global data that you need access to for this ABL Session
 */
};

int proHLC_InitSession()
{
    void *ptr = promalloc(sizeof(struct myglobaldata));
    /* Initialize myglobaldata structure with your non-constant
     * global data that you need access to for this ABL Session
     */
    proSetGlobalStruct(ptr); /* stash it so we can get it later */
}

int proHLC_ExitSession()
{
    struct myglobaldata *ptr = (void *)proGetGlobalStruct();
    /* First deallocate any allocated memory
     * referenced from myglobaldata */
    if (ptr)
```

```
        profree((void *)ptr); /* free up the structure itself */
    proSetGlobalStruct((void *) 0); /* so we don't dereference */
}
```

The `hldemo.p` example is the front end to the sample test application.

hldemo.p

```
DEFINE VARIABLE answer AS CHARACTER NO-UNDO FORMAT "9".
DEFINE VARIABLE choice AS CHARACTER NO-UNDO INITIAL "1,2,3,4".
DEFINE VARIABLE last1  AS LOGICAL   NO-UNDO INITIAL TRUE.
DEFINE VARIABLE last2  AS LOGICAL   NO-UNDO INITIAL TRUE.

RUN loaddb.p.

INPUT FROM TERMINAL.
OUTPUT TO TERMINAL.

FORM SKIP(4) SPACE(4)
  " 1 - Installation Test" SKIP(1) SPACE(4)
  " 2 - TANK Capacity Calculation demo " SKIP(1) SPACE(4)
  " 3 - Screen raw/cooked demo " SKIP(1) SPACE(4)
  " 4 - Exit Session" SKIP(2) SPACE(4)
  " Enter example selection ===> " answer no-label
  WITH CENTERED TITLE " EXAMPLE MENU " FRAME example.

FORM last1 last2 WITH FRAME lastcheck.

REPEAT:
  HIDE ALL.
  UPDATE answer AUTO-RETURN
  VALIDATE(LOOKUP(answer,choice) <> 0,"Enter one of choices displayed.")
  WITH FRAME example.
  HIDE ALL.
  IF answer = "1" THEN DO:
    /* Check to see if need to reinitialize database. */
    IF last1 THEN DO:
      last1 = TRUE.
      last2 = FALSE.
      RUN initdb.p.
      HIDE ALL.
    END.
    last1 = TRUE.
    RUN hltest.p.
  END.
  ELSE IF answer = "2" THEN DO:
    /* Check to see if need to reinitialize database. */
    IF last2 THEN DO:
      last2 = TRUE.
      last1 = FALSE.
      RUN initdb.p.
      HIDE ALL.
    END.
    last2 = TRUE.
    RUN hltank.p.
  END.
  ELSE IF answer = "3" THEN RUN hlscreen.p.
  ELSE IF answer = "4" THEN QUIT.
end.
HIDE ALL.
QUIT.
```

10

Introduction to External Program Interfaces

These topics introduce the OpenEdge *external program interfaces* (EPIs). These consist of ABL (Advanced Business Language) statements and supporting software that enable an ABL application to exchange data and services with external non-ABL applications. To use these interfaces, you must be proficient in the programming tools, applications, and environments you want to access from ABL.

For details, see the following topics:

- Using MEMPTR to reference external data
- System clipboard
- Named pipes
- UNIX shared library and Windows DLL support
- Windows Dynamic Data Exchange
- COM objects: Automation objects and ActiveX controls
- Sockets

Using MEMPTR to reference external data

The ABL `MEMPTR` data type provides a means to access (*unmarshal* and *marshal*) data that you plan to pass to and from other application and system environments using the following EPIs:

- UNIX shared object libraries and Windows dynamic link libraries (DLLs)
- Sockets

- XML

A MEMPTR variable references a region of memory set to a size that you specify. Once you declare and initialize the MEMPTR variable to the specified size, you can read and modify the memory region it references as follows:

- Read and write data values to specified locations within the MEMPTR region. These values can contain different numbers of bytes, depending on a data type that you specify for the value read or written.
- Read and write individual bits within an ABL INTEGER variable, whose value you can also read and write to a MEMPTR region.
- Assign values between two MEMPTR variables and between a MEMPTR variable and a RAW variable (or field).
- Indicate the byte order of data you have written so that a target system can interpret the MEMPTR data appropriately. This is primarily useful for exchanging MEMPTR data between different system environments using sockets.

Together, these features allow you to store and access data aggregates (*structures*), including complete OpenEdge database records.

The following sections describe the general capabilities of MEMPTR data, including:

- Comparing MEMPTR and RAW data types
- Initializing and uninitializing MEMPTR variables
- Reading and writing data in MEMPTR variables
- Retrieving and storing pointers
- Setting byte order

For more information on using MEMPTR for a specific EPI, see:

- UNIX shared library and Windows DLL support on page 424, and Shared Library and DLL Support on page 483
- Sockets on page 432, and Sockets on page 579

Comparing MEMPTR and RAW data types

ABL provides two data types to store data in a raw (binary) form, MEMPTR and RAW. MEMPTR and RAW data types provide similar features, but serve different purposes, as follows:

- You can use both MEMPTR and RAW variables to store binary data that originates with any other data type in ABL or that you import from another environment using an EPI.
- You can use MEMPTR variables to directly exchange data between ABL and a supported EPI environment; you can use RAW variables only to exchange data with another ABL data type, including MEMPTR.
- You can define RAW fields in an OpenEdge database, but you cannot define MEMPTR fields. You can move database fields and records to and from MEMPTR variables using RAW variables as intermediate storage.

In sum, the RAW data type supports the storage and movement of binary data within the local ABL and OpenEdge database environment, while the MEMPTR data type supports the storage and movement of data between the local ABL and external ABL or non-ABL environments.

You can also assign values between MEMPTR and RAW variables directly using the ABL assignment operator (=). If the target variable is RAW, ABL resizes the target, if necessary, to make it the same size as the source. For both assignment between MEMPTR and RAW variables and assignment between two variables of the same data type, the source and the target variables maintain separate and complete copies of the data.

Initializing and uninitializing MEMPTR variables

After declaring a MEMPTR variable, you must initialize it before you can use it. In general, to initialize a MEMPTR variable, you use the SET-SIZE statement to allocate a region of memory and associate it with the variable.

Note: You can also allocate the memory region for a MEMPTR variable using a shared library routine. In this case, you must still use the SET-SIZE statement to initialize the size of the returned memory region in ABL.

Initializing MEMPTR variables using the SET-SIZE statement

This is the syntax for the SET-SIZE statement:

Syntax

```
SET-SIZE ( mptr-name ) = expression
```

The value *mptr-name* is the name of a MEMPTR variable, and *expression* is an integer expression specifying the size, in bytes, of a memory region associated with *mptr-name*. The value of *expression* can also be zero (0), which frees any memory previously allocated to *mptr-name*. ABL uses this size to perform bounds checking that ensures you do not read or write to portions of memory outside of the specified region.

If *mptr-name* is uninitialized (that is, defined but not associated with any memory region) and *expression* is greater than zero (0), the SET-SIZE statement allocates a memory region whose size is specified by *expression*, and associates it with *mptr-name*. If *mptr-name* is already initialized and *expression* is greater than zero (0), the SET-SIZE statement has no effect.

To resize memory allocated to a MEMPTR variable, you must invoke SET-SIZE with an *expression* of zero (0), then invoke SET-SIZE with an *expression* equal to the new allocation.

Checking a MEMPTR variable for initialization

You might not be sure whether a particular MEMPTR variable is already initialized when you try to initialize it. You can verify that the variable is initialized and obtain the size of its memory region using the GET-SIZE function. This is the syntax for the GET-SIZE function:

Syntax

```
GET-SIZE ( mptr-name )
```

If *mptr-name* is initialized, the GET-SIZE function returns the size of its memory region. Otherwise, it returns zero (0).

Note: To return a memory size greater than zero (0), the variable must be fully initialized with the SET-SIZE statement, not just allocated by a shared library routine.

Freeing memory associated with a MEMPTR variable

The region of memory associated with a MEMPTR variable remains allocated until it is freed. ABL does not automatically free the memory for you. It is up to you to ensure that the memory is freed.

You can free the memory using SET-SIZE with an *expression* of zero (0). SET-SIZE then deallocates (frees) any memory region associated with *mptr-name*. This makes *mptr-name* uninitialized.

Note: You must fully understand the memory management provided by shared library routines before you can effectively using MEMPTR variables with them.

Reading and writing data

Once you have initialized a MEMPTR variable, you can build a data aggregate (structure) or access an existing structure in the associated memory region using several memory-writing statements and memory-reading functions. Memory-writing statements write values to specified locations in the memory region. Memory-reading functions return values from the specified locations in the memory region. Through an appropriate choice of these statements and functions you can thus copy ABL data types and bit fields to and from MEMPTR variables. You can also copy complete OpenEdge database records to and from MEMPTR variables.

Note: Before setting or getting values in a MEMPTR variable, you might want to check the MEMPTR size using the GET-SIZE function. For more information, see Initializing and uninitializing MEMPTR variables on page 415.

This is the syntax for the MEMPTR memory-writing statements (except PUT-STRING):

Syntax

```
PUT-datatype ( mptr-name , byte-position ) = expression
```

This is the syntax for the MEMPTR memory-reading functions (except GET-STRING and GET-BYTES):

Syntax

```
GET-datatype ( mptr-name , byte-position )
```

Each PUT-*datatype* statement writes a value (*expression*) of a certain data type to the memory region associated with the MEMPTR variable *mptr-name* at the specified *byte-position*. Each GET-*datatype* function reads and returns the value of a data type from the memory region associated with the MEMPTR variable *mptr-name* at the specified *byte-position*. The *byte-position* in these statements and functions is specified by an integer expression that starts at one (1).

The PUT-STRING statement and GET-STRING function each allow an additional optional parameter that specifies the number of bytes to write or read in the MEMPTR variable. The GET BYTES function has the same parameter, but it is required for this function.

Note: The *mptr-name* parameter in these statements and functions can reference RAW as well as MEMPTR variables. However, the EPIs that require MEMPTR variables cannot use RAW variables directly. You must convert RAW values to MEMPTR before using them with these EPIs. You can do this using direct assignment between RAW and MEMPTR variables or by using statements and functions such as PUT-BYTES, GET-BYTES, or GET-RAW.

Memory-writing statements

ABL provides the PUT-*datatype* statements shown in the following table.

Table 46: PUT-datatype statements

Statement	Description
PUT-BITS	Uses the bit representation of an integer to set a given number of bits at a given location within another integer
PUT-BYTE	Writes an INTEGER value to the specified 1-byte location
PUT-BYTES	Writes the contents of a RAW or MEMPTR variable to the specified byte location of a RAW or MEMPTR variable
PUT-SHORT	Writes an INTEGER value to the specified 2-byte location
PUT-UNSIGNED-SHORT	Writes an unsigned INTEGER value to the specified 2-byte location
PUT-LONG	Writes an INTEGER or INT64 value to the specified 4-byte location
PUT-UNSIGNED-LONG	Writes an unsigned INTEGER or INT64 value to the specified 4-byte location
PUT-INT64	Writes an INTEGER or INT64 value to the specified 8-byte location
PUT-FLOAT	Writes a decimal value to the specified 4-byte location as a single-precision floating-point value
PUT-DOUBLE	Writes a decimal value to the specified 8-byte location as a double-precision floating-point value
PUT-STRING	Writes a character string value to the specified location, either null terminated or for a specified number of bytes

For more information on these statements, see *ABL Reference*.

Memory-reading functions

ABL provides the GET-*datatype* functions shown in the following table.

Table 47: GET-datatype functions

Function	Description
GET-BYTE	Returns the INTEGER value of the specified 1-byte location
GET-SHORT	Returns the INTEGER value of the specified 2-byte location

Function	Description
GET-UNSIGNED-SHORT	Returns the value of the specified 2-byte location, interpreted as an unsigned INTEGER
GET-UNSIGNED-LONG	Returns the unsigned 32-bit value at the specified memory location as an INT64
GET-INT64	Returns the signed 64-bit value at the specified memory location as an INT64 value. The specified memory location can have a RAW or a MEMPTR value
GET-LONG	Returns the INTEGER value of the specified 4-byte location
GET-FLOAT	Returns the DECIMAL value of the specified 4-byte location, interpreted as a single-precision floating-point value
GET-DOUBLE	Returns the DECIMAL value of the specified 8-byte location, interpreted as a double-precision floating-point value
GET-STRING	Returns the CHARACTER string value from the specified location, either null terminated or for a specified number of bytes that can include nulls
GET-BYTES	Returns, as a MEMPTR or RAW value, the specified number of bytes from the specified byte location of a RAW or MEMPTR variable

For more information on these functions, see *ABL Reference*.

Copying between basic ABL data types and MEMPTR

The following table lists the basic ABL data types and how you can copy them in and out of a MEMPTR variable.

Table 48: Copying between basic ABL data types and MEMPTR

Data type	Copying to/from MEMPTR
DATE	To copy into a MEMPTR: • Use INTEGER(*date-expression*) to obtain an integer value • Use the PUT-LONG statement to copy integer value to MEMPTR To copy from a MEMPTR: • Use the GET-LONG function to return the integer value • Use the DATE(*integer-expression*) function to return the date

Data type	Copying to/from MEMPTR
DECIMAL[9]	To copy into a MEMPTR, use the PUT-DOUBLE or PUT-FLOAT statement. To copy from a MEMPTR, use the GET-DOUBLE or GET-FLOAT function.
INTEGER[9]	To copy into a MEMPTR, use the PUT-LONG, PUT-SHORT, PUT-UNSIGNED-SHORT, or PUT-BYTE statements. To copy from a MEMPTR, use the GET-LONG, GET-SHORT, GET-UNSIGNED-SHORT, or GET-BYTE functions.
LOGICAL[9]	To copy into a MEMPTR, use the PUT-LONG, PUT-SHORT, PUT-UNSIGNED-SHORT, or PUT-BYTE statements. To copy from a MEMPTR, use the GET-LONG, GET-SHORT, GET-UNSIGNED-SHORT, or GET-BYTE functions.
RAW	To copy into a MEMPTR, assign the RAW value directly using an assignment statement or use the PUT-RAW statement. To copy from a MEMPTR, assign the MEMPTR value directly using an assignment statement or use the GET-RAW function.

Manipulating bit values

You can copy bit values up to the size of an ABL INTEGER from one INTEGER value to another. The statement to copy bit values to an INTEGER variable, PUT-BITS, has the following syntax:

Syntax

```
PUT-BITS( destination , start-position , count ) = integer-expression
```

This statement interprets an integer (*integer-expression*) as the sequence of bits representing the binary value of *integer-expression*. For example, if the value of *integer-expression* is 22, the bit sequence is 10110. The statement interprets the INTEGER variable, *destination*, as a sequence of bits and writes the sequence of bits from *integer-expression* into *destination*, starting at the specified bit position (*start-position*). The bit position in *destination* is counted from the low-order bit, where the first bit is bit one (1). If the value of *integer-expression* is too large to store in the specified number of bits, ABL stores the low-order *count* bits of *integer-expression* in the specified *count* bits within *destination*.

The function to return some number of bits from an INTEGER variable, GET-BITS, has the following syntax:

Syntax

```
GET-BITS( source , start-position , count )
```

[9] The choice of exact statement or function to use depends on the data type used by the shared library routine or the socket application with your application is communicating.

This function returns the INTEGER that represents the value of the number of bits specified by *count* starting at the specified low-order bit position (*start-position*) within the INTEGER variable specified by *source*.

Thus, you can store bit values in MEMPTR variables and return bit values from MEMPTR variables by using the PUT-LONG statement and GET-LONG function to store and return the corresponding INTEGER expression that contains the bit pattern.

Copying between database records and MEMPTR

Copying a database record to and from a MEMPTR variable relies on the RAW-TRANSFER statement. This statement allows you to copy a whole database record buffer to a RAW variable or to copy a RAW variable to a database record buffer.

To store a database record in a **MEMPTR** variable:

1. Copy the record buffer to a RAW variable using the RAW-TRANSFER statement.
2. Assign the RAW variable to the MEMPTR variable or to the specified position in the MEMPTR variable using the PUT-BYTES statement.

To retrieve a database record from a **MEMPTR** variable:

1. Assign the MEMPTR variable to a RAW variable or use the GET-BYTES function to copy the specified bytes from the MEMPTR to the RAW variable.
2. Copy the RAW variable to the record buffer using the RAW-TRANSFER statement.

Retrieving and storing pointers

In some cases (especially for shared library routines), you might have to obtain a pointer to the memory region associated with a MEMPTR variable. You might need this value, for example, to build a structure that contains a pointer to another structure. You can get a pointer to a MEMPTR region by using the GET-POINTER-VALUE function. This is the syntax for the GET-POINTER-VALUE function:

Syntax

```
GET-POINTER-VALUE ( memptr-name )
```

For example, to retrieve the pointer to a memory region specified by a MEMPTR variable (BitMapInfo) and store it in the first byte position of another MEMPTR variable (BitMaps), you can use the following statement (assuming 32-bit pointers):

```
PUT-LONG(BitMaps,1) = GET-POINTER-VALUE(BitMapInfo).
```

You can also store the address of a memory item into a MEMPTR variable by using the SET-POINTER-VALUE statement. Here is the syntax:

Syntax

```
SET-POINTER-VALUE ( memptr-name ) = address
```

For example, to store an address located at the second byte of a MEMPTR variable (BitMaps) into another MEMPTR variable (BitMapInfo), you can use the following statement:

```
SET-POINTER-VALUE(BitMapInfo) = GET-LONG(BitMaps,2)
```

For more information on the GET-POINTER-VALUE function and the SET-POINTER-VALUE statement, see *ABL Reference*.

Setting byte order

When passing MEMPTR data between system environments (such as can happen using sockets), you must help ensure that the communicating applications agree on the byte order for any data types that are transferred using the MEMPTR variable. Most machines follow one of two schemes for ordering bytes:

- **Little-endian** — Low-order bytes are stored starting at the lowest address location reserved for a data type, with progressively higher-order bytes stored at the higher byte positions. For example, 7 is stored in a four-byte (hex) integer as follows, where addresses increase from left to right:

  ```
  07 00 00 00
  ```

- **Big-endian** — High-order bytes are stored in the lowest address location reserved for a data type with progressively lower-order bytes stored at the higher byte positions. For example, 7 is stored in a four-byte (hex) integer as follows, where addresses increase from left to right:

  ```
  00 00 00 07
  ```

ABL ensures that MEMPTR data is interpreted correctly using MEMPTR write/read statements and functions as long as you indicate what byte order the MEMPTR data that you write uses. To do this use the SET-BYTE-ORDER statement:

Syntax

```
SET-BYTE-ORDER( memptr-name ) = integer-expression
```

Given the name of the MEMPTR variable (*memptr-name*), you can set one of these keyword values for *integer-expression*:

- **HOST-BYTE-ORDER** — A value of one (1) that indicates the byte ordering of the machine where the statement executes
- **BIG-ENDIAN** — A value of two (2) that indicates big-endian byte ordering

  Note: Internet protocols use big-endian byte ordering.

- **LITTLE-ENDIAN** — A value of three (3) that indicates little-endian byte ordering

By default, all MEMPTR variables use the byte ordering of the machine where they are defined.

Note that the SET-BYTE-ORDER statement does not change the existing order of bytes in the MEMPTR data. It only indicates how subsequent MEMPTR write/read statements and functions interpret the data. You can also return the current byte order setting using the GET-BYTE-ORDER function. For more information, see *ABL Reference*.

Defining a MEMPTR with Memory Stream classes

The memory stream classes, Progress.IO.MemoryOutputStream and Progress.IO.MemoryInputStream encapsulate the MEMPTR data type, and make it easier to perform MEMPTR operations.

For example, you can write to memory referenced by a MEMPTR without worrying about resizing, keeping track of read/write positions, or remembering when to free it. The memory stream object automatically resizes the MEMPTR structure when needed and frees it when the stream object goes out of scope.

Note: The memory stream object's internal MEMPTR property is read-only. It can be publicly accessed for read operations such as GET-STRING, GET-BYTES, GET-SIZE and so on. Any write operations such as PUT-STRING, PUT-BYTE, or SET-SIZE on this property causes a runtime error indicating that AVM is unable to perform such operation on a read only MEMPTR.

The following ABL code samples demonstrate how memory stream classes simplify the use of MEMPTRs.

MEMPTRs defined without using memory stream classes

```
1.   DEFINE VARIABLE len          AS INT64.
2.   DEFINE VARIABLE i            AS INT64.
3.   DEFINE VARIABLE mData        AS MEMPTR.
4.   DEFINE VARIABLE iPosition    AS INT.
5.   DEFINE VARIABLE iLen         AS INT.
6.   DEFINE VARIABLE tempM        AS MEMPTR.
7.   DEFINE VARIABLE size         AS INT64.
8.   size = 1000.
9.   SET-SIZE(mData) = size.
10.  iPosition = 1.
11.  FOR EACH CUSTOMER:
12.     iLen = LENGTH(Name).
13.     IF iPosition + iLen > size THEN DO:
14.       tempM = mData.
15.       size = size * 2.
16.       SET-SIZE(mData) = 0.
17.       SET-SIZE(mData) = size.
18.       PUT-BYTES(mData,1) = tempM.
19.       SET-SIZE(tempM) = 0.
20.     END.
21.     PUT-STRING(mData, iPosition, iLen) = Name.
22.     iPosition = iPosition + iLen.
23.  END.
24.  . . . /* Use memptr, for example send on a socket connection */
25.  SET-SIZE(mData) = 0.
```

MEMPTRs defined as memory stream classes

```
1.   USING Progress.IO.
2.   DEFINE VARIABLE mStreamObj AS Progress.io.MemoryOutputStream.
3.   mStreamObj = NEW Progress.io.MemoryOutputStream().
4.   FOR EACH CUSTOMER:
5.     mStreamObj:Write(Name).
6.   END.
7.   /* mStreamObj goes out of scope, gets garbage collected, and memory
8.   is automatically freed. */
```

See the *ABL Reference* for more information about these memory stream classes.

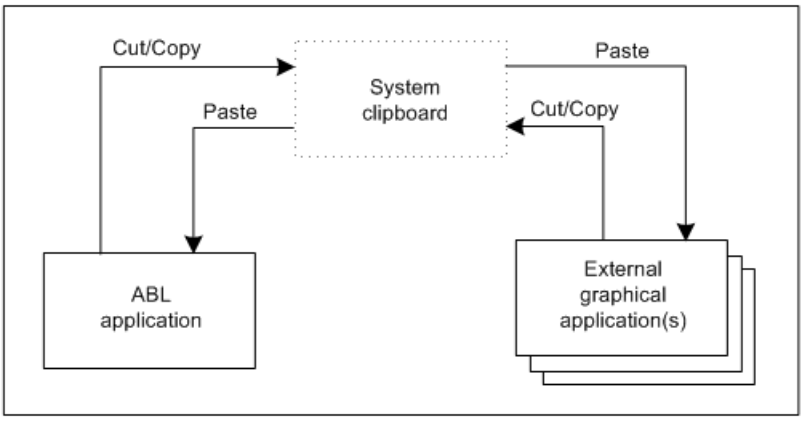
System clipboard

The *system clipboard* allows a window system user to transfer data between one widget (or application) and another using cut, copy, and paste operations. Each window system provides its own style of clipboard support. However, it generally allows the user to perform these clipboard operations in exactly the same way between all applications on the system using some combination of the mouse and keyboard.

The system clipboard and ABL

The following figure shows the typical interactions between an ABL application and other graphical applications using the clipboard.

Figure 33: ABL interactions using the system clipboard



The dotted box emphasizes that, from the user's viewpoint, the clipboard is essentially invisible.

ABL supports clipboard interactions with other graphical applications in the Windows environments. In character mode, ABL supports clipboard interactions between fields in a single OpenEdge application. Using the `CLIPBOARD` system handle and an appropriate set of user-interface triggers, you can define the response of your OpenEdge application to the standard clipboard operations on your system. For example, you can eliminate cut operations (that remove data) and provide only copy and paste; or you can specify how the data is transferred, whether all or part of it is transferred, and where it goes when it is pasted into your OpenEdge application.

Requirements for using the CLIPBOARD handle

The minimum requirement for using the `CLIPBOARD` handle is a thorough knowledge of how the clipboard operates on your operating system from the user's viewpoint. You can then use the `CLIPBOARD` handle to provide or modify that functionality in your OpenEdge applications.

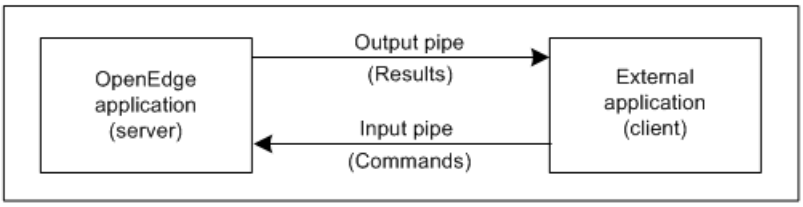
Named pipes

Named pipes provide the basic interprocess communications (IPC) mechanism in the UNIX environment. This IPC mechanism allows two processes on UNIX to send data to each other as if they are reading and writing sequential files. Each process opens a separate pipe for input or output to the other, and the input process waits for the output process to send data each time it reads its input pipe. Named pipes are also supported in the Windows environment and generally behave similarly to UNIX named pipes.

Named pipes and ABL

The following figure shows typical data exchanges between an OpenEdge application and an external application using named pipes.

Figure 34: ABL exchanging data using named pipes



A OpenEdge database client accesses named pipes in the same manner as any input or output file. Typically, the OpenEdge database client acts as an ABL server, receiving ABL requests from a non-OpenEdge UNIX or Windows application through an input pipe, executing the ABL statements, and returning the results to the UNIX or Windows application through an output pipe. Named Pipes on page 465 emphasizes this type of client/server interaction with ABL using named pipes.

Requirements for using named pipes

The minimum requirement for working with named pipes in ABL is a working knowledge of the UNIX operating system and its shell commands or of the Windows operating system, and of how the non-OpenEdge application you want to communicate with uses pipes. Proficiency in the C programming language is also helpful when working with the named pipe examples in this manual.

UNIX shared library and Windows DLL support

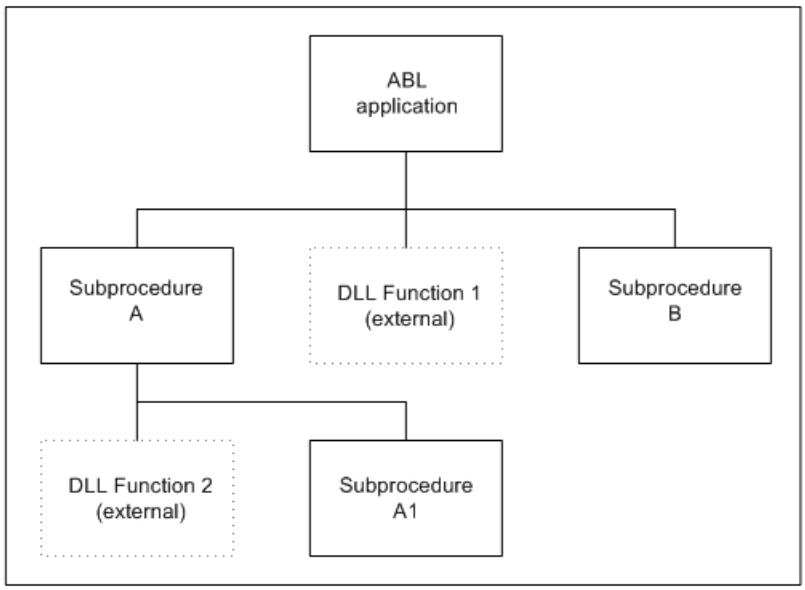
The UNIX and Windows operating systems support the use of an executable file that contains compiled functions or routines that can be linked to an application at runtime rather than at build time. This executable file is called a *shared object* or *shared library* on UNIX and a *dynamic link library* (DLL) in Windows. This dynamic linking capability promotes a building block approach to application development; third-party software packages provide much of their functionality in shared libraries. It also makes function upgrades more easily available to applications that use them.

Throughout this section, the term "shared library" is used when a statement is applicable to both the UNIX and Windows systems. "UNIX shared library" is used if a statement is applicable only to UNIX; "DLL" is used for statements applicable only to Windows.

Shared libraries and ABL

The following figure shows a top-down structure diagram for an OpenEdge application that calls shared library functions.

Figure 35: OpenEdge application calling DLL functions



Note: Both OpenEdge graphical and character applications can access DLLs in Windows.

The dotted boxes emphasize that although shared library functions are called from within an OpenEdge application, they actually reside in external shared libraries.

An OpenEdge application can access a shared library function much like an internal ABL procedure. The client declares the function and its file using a `PROCEDURE` statement, defining all function parameters with `DEFINE PARAMETER` statements. ABL provides a special set of parameter data types to match C and Windows data types, and a set of statements and functions to build and access C structures from ABL. The client executes the shared library function exactly like an ABL procedure, using the `RUN` statement.

Requirements for using shared libraries

The minimum requirement for working with shared libraries in ABL is knowledge of how to use the function parameters. Declaring shared library functions in ABL also requires a basic knowledge of C programming and of the operating system(s) on which your applications run.

It might be helpful for at least one programmer proficient in Windows or UNIX and C to provide the required shared library function declarations in include files. They might also want to write procedures or include files that appropriately create, set, and read any C structures used by the functions. All other ABL programmers only have to use the procedures and include files provided by this programmer. Otherwise, they execute the shared library functions like any ABL procedure.

For more information about accessing and executing Windows DLL routines and UNIX shared library routines, see Shared Library and DLL Support on page 483.

Windows Dynamic Data Exchange

Dynamic data exchange (DDE) is a protocol Windows provides for inter-process communications.

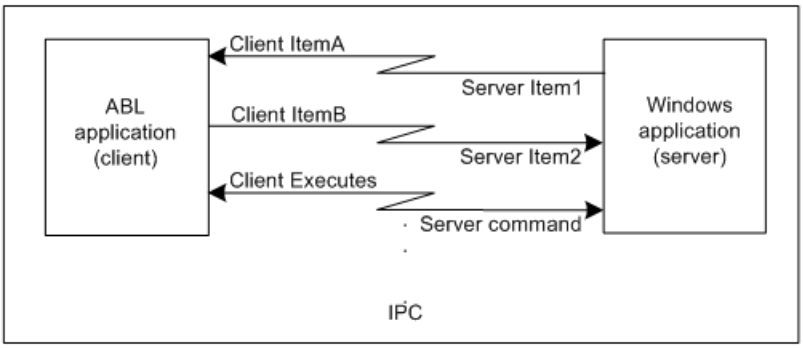
Note: OpenEdge support for DDE is a deprecated feature. OpenEdge supports DDE for backward compatibility only. For inter-process communication on Windows platforms, consider using the Microsoft Component Object Model (COM). OpenEdge support for COM is documented in Using COM Objects in ABL on page 523.

Using this protocol, two applications communicate in a client/server relationship in which the client initiates the communications and the server exchanges data with and provides services to the client. This is a very flexible IPC mechanism that enables a range of capabilities from simple data transmission between two applications to the ability for multiple applications to "work" in each other's environments. For example, a word processing application might create and modify spreadsheets in a spreadsheet application, and the spreadsheet application might, in turn, create and modify documents in the word processing application.

DDE and ABL

The following figure shows a series of IPC exchanges between an OpenEdge application and another Windows application using DDE.

Figure 36: ABL exchanging data using DDE



Note: Both OpenEdge graphical and character applications can use DDE in Windows.

In this example, the OpenEdge application sets the value of ABL ItemA from Server Item1; sets the value of Server Item2 from ABL ItemB; and executes a command on the Server, possibly returning a data value or error condition.

ABL supports DDE as a client only. This allows OpenEdge database clients to communicate with any other Windows application with DDE server capability. Examples of Windows applications with DDE server capability include the Windows Program Manager, Microsoft Excel for Windows, and Visual Basic applications. As a DDE client, ABL can, for example, create and modify worksheets in Excel, and at the same time automatically receive notification of updates to worksheet cells (data items) from Excel. Excel, as the DDE server, provides this notification by sending an event that ABL can handle in a trigger.

Requirements for using DDE

The minimum requirement for working with DDE in ABL is familiarity with the way your DDE server applications provide data and services to DDE clients. Each server application has its own way of naming data items and groups of data items in its own environment. It also usually provides a set of commands that you can execute with DDE that direct the server to perform server-supported application actions (for example, to create documents or spreadsheets). For information on using DDE with each application, see the application documentation.

COM objects: Automation objects and ActiveX controls

COM objects are encapsulated Windows application objects that conform to specifications of the Microsoft Component Object Model. As such, COM objects provide functionality for an application that might not otherwise be supported by ABL. This allows you to acquire functional elements for your OpenEdge applications from third-party vendors as well as from Progress Software Corporation.

The COM standard allows ABL to access a COM object through its properties, methods, and events. This is analogous to how ABL provides access to widget attributes, methods, and events. However, ABL provides access to COM objects through an industry-standard mechanism and provides access to widgets through a proprietary mechanism.

COM objects supported in ABL

ABL supports two classes of COM objects:

- ActiveX Automation objects
- ActiveX controls

ActiveX Automation objects

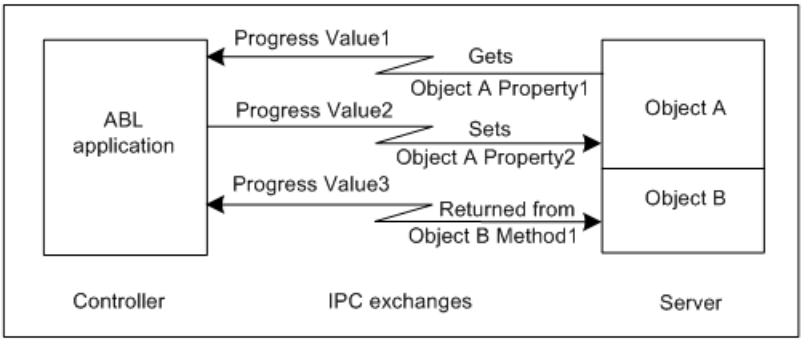
ActiveX Automation objects (or just *Automation objects*) are COM objects that encapsulate all or part of an application in a stand-alone executable (EXE file) or dynamic link library (DLL) file. These Automation objects make the encapsulated functionality available to another application.

Similar to DDE, your OpenEdge application functions in a client/server relationship to the application that provides an Automation object. As such, ABL functions as an ActiveX Automation Controller and accesses the Automation object in the application that functions as an ActiveX Automation Server.

For example, your OpenEdge application reads and writes data values in the Automation Server by accessing properties and methods of an available Automation object, such as a spread sheet or word-processing document. Depending on the server application, you might even be able to create new Automation object instances (new spread sheets or documents) in the Automation Server from within your OpenEdge application.

The following figure shows a series of IPC exchanges between an OpenEdge application and the Automation objects of an Automation Server:

Figure 37: ABL using ActiveX Automation



Note: Both ABL graphical and character applications can use ActiveX Automation in Windows.

In this example, the OpenEdge application gets ABL Value1 from Object A Property1; sets Object A Property2 from ABL Value2; and gets ABL Value3 from a call to Method1 of Object B, possibly after creating an instance of Object B.

Compare this functionality with DDE (see Figure 36: ABL exchanging data using DDE on page 426). The effects are very similar. However, the access to properties and methods provided by the component model of ActiveX Automation is much more straightforward, robust, and flexible.

ActiveX controls

ActiveX controls (OCXs) are COM objects that rely on COM standards, including ActiveX Automation, to communicate with an application and also to provide a mechanism to generate events. ActiveX controls reside only in DLL files that provide the complete implementation, which often includes a user-interface component. As such, ActiveX controls are directly analogous to ABL widgets, but often include a variety of capabilities not available with the widgets built into ABL. For example, you can find ActiveX controls that function as calendars, pie charts, bar graphs, gauges, meters, and even communications, timing, and parsing controls that have no user-interface components.

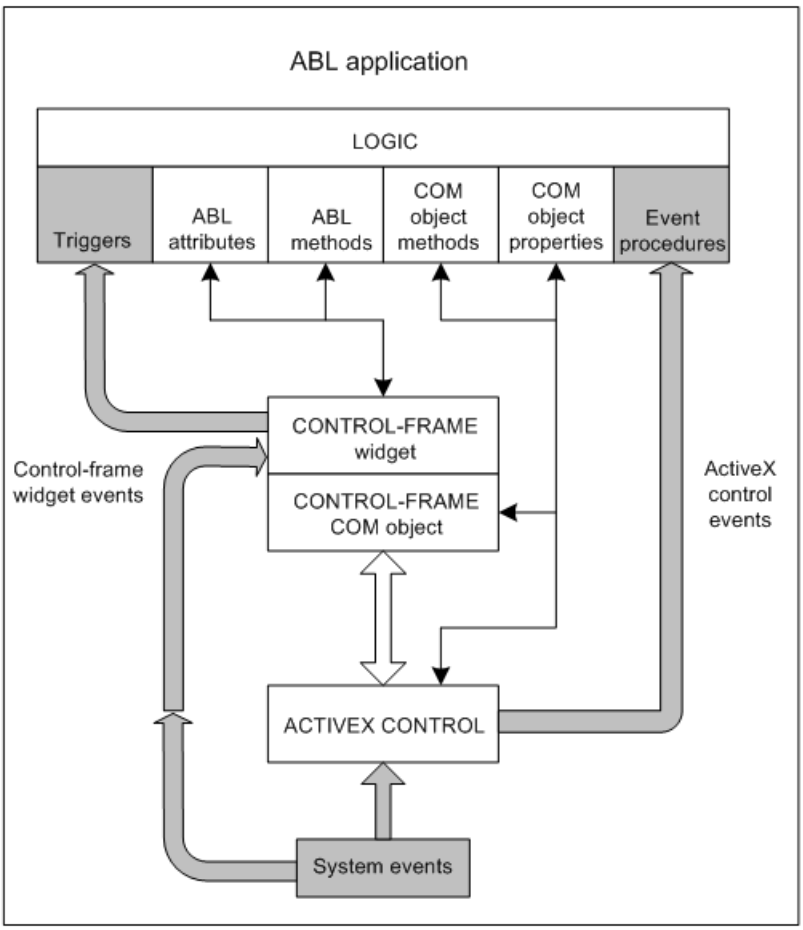
Whether or not an ActiveX control includes a user-interface component, ABL supports a user-interface widget, the `CONTROL-FRAME` widget, to make the control available to your application. The control-frame widget anchors the ActiveX control to your application. This widget physically orients the control in the ABL user interface, but provides no other services for accessing it. A separate but related COM object, the control-frame COM object, provides the real control container support. This special ABL-supported COM object contains and provides direct access to the ActiveX control from ABL.

Note: Unlike with ActiveX Automation, you can use ActiveX controls only in OpenEdge graphical applications.

Support for VBX controls in earlier versions of Progress requires you to use special methods on a control-container widget to access any VBX control property or method. However for ActiveX controls, ABL allows you to access control properties and methods directly, without reference to the control-frame widgets that anchor them.

The following figure shows how ABL supports ActiveX controls.

Figure 38: ABL interface to ActiveX Controls



The control-frame widget and COM object work together to connect the ActiveX control to your application. As you might suspect, ABL supports different types of handles for widgets and COM objects. Thus, you access control-frame widgets using widget handles and access all COM objects (including ActiveX controls) using component handles. It is these component handles that allow you to gain direct access to the properties and methods of an ActiveX control (or Automation object).

Similar to VBX control support in earlier versions, ABL allows you to define event procedures to handle events generated directly by ActiveX controls. You can also handle events on the control-frame widget with user-interface triggers. Control-frame widget events (like TAB or LEAVE) allow you to coordinate user-interface actions between ActiveX controls and ABL widgets.

Support for COM object properties and methods

In ABL, the first step to access a COM object is to obtain its component handle. Once you have the component handle, you can use it to access properties and methods supported by the COM object. To support these property and method references, ABL also provides automatic mappings between COM data types and ABL data types. This allows you to pass COM object property, method, and event parameter values directly as ABL data items without the need for data conversion functions.

For more information on accessing COM object properties and methods, see Using COM Objects in ABL on page 523.

Support for Automation object events

Automation objects can generate events in response to an action performed on the object, such as the creation of a Word document or the printing of an Excel Workbook. ABL supports event notification for ActiveX Automation objects using a built-in method on the COM object, ENABLE-EVENTS. Once events are enabled, ABL searches for a running or persistent procedure matching the event name. For more information on handling events for Automation objects, see ActiveX Automation Support on page 543.

Support for ActiveX control events

ActiveX controls respond to events much like ABL widgets. ActiveX controls that provide a user interface typically generate user-interface events, similar to user-interface widgets. However, ActiveX controls can support other types of functionality that generate other types of events, such as events that notify the arrival of a message for a communications application or the change in temperature of a manufacturing process for a data acquisition application.

Also, unlike widget events, ActiveX control events can pass parameters like a procedure call. Thus, ABL provides a type of internal procedure (*OCX event procedure*) to handle ActiveX control events. You can handle any ActiveX control event using an OCX event procedure.

ABL also allows you to handle certain events on the control-frame widget instead of on the ActiveX control. When ActiveX controls have focus, they generally take over the input and your application receives most events directly as ActiveX control events. However, when an ActiveX control has no equivalent event or when it is necessary to manage the orientation of the control in the ABL user interface, you can handle some input actions as field-level widget events on the control-frame. You can handle these widget events using the standard ABL ON statement, but ABL executes only one event handler (an ABL trigger or an OCX event procedure) per event.

For more information on handling events for ActiveX controls, see ActiveX Control Support on page 551.

COM object sources

Any number of Automation objects and ActiveX controls might be available on your Windows system, depending on its configuration and the applications you have installed. In general, you can get information on the available COM objects by using the COM Object Viewer that comes installed with ABL. For more information on this viewer, see Using COM Objects in ABL on page 523.

Automation objects

Since Automation objects are generally part of a stand-alone application that functions as an Automation Server, you must have the application installed that provides the Automation objects that you need for your OpenEdge application. For information on the available Automation objects in a particular Automation Server application, see the documentation for that application or use the OpenEdge COM Object Viewer.

ActiveX controls

Three ActiveX controls come installed with OpenEdge for Windows. These include a combo box, a spin button, and a timer control. For more information on these controls, see ActiveX Control Support on page 551.

Visual Basic Professional provides a starter set of ActiveX controls and includes basic documentation on working with ActiveX controls. Other commercial vendors and countless sources of shareware and freeware offer ActiveX control packages of varying quality.

Caution: The control vendor is responsible for following COM standards. Any deviation might result in a control that does not work in the OpenEdge environment.

Requirements for using Automation objects

The main requirement for using an Automation object is that the Automation Server application must be installed on your system and ready to execute. For more information, see the documentation on ActiveX Automation support that comes with your Automation Server application.

Requirements for using ActiveX controls

You can access ActiveX controls in two modes:

- Design mode
- Run mode

Design-mode (or design-time) access allows you to modify properties that initialize the control and define it for use in an application. For many controls, these properties affect such attributes as color and size, but also enable and disable other special features unique to each ActiveX control. In general, design-time properties affect the appearance and initial internal state of an ActiveX control. Design-time properties are generally readable, but might not be writable at runtime.

Run-mode (or runtime) access allows your application to interact with the control, responding to events, invoking methods, and getting and setting properties that affect the ActiveX control at runtime. Run-time properties might not be readable or writable at design time.

The requirements for working in each mode differ.

Design-mode requirements

To define an ActiveX control for use in an OpenEdge application, you must:

- **Have the ActiveX control installed in your Windows environment** — This includes one or more DLL files. Install the ActiveX control according to the vendor's instructions. The primary control file usually has an `.ocx` extension. Note that other files might also be required and installed, as well.
- **Have a license installed that allows you to access the ActiveX control in design mode** — Not all ActiveX controls require licenses for design-time access, but most commercial ActiveX controls do. Licenses generally come with the vendor's installation, and are either recorded in the registry or in a license file, often with a `.lic` extension. ABL stores the license information for the ActiveX controls that it installs in the registry.
- **Use the OpenEdge AppBuilder to create an instance of the ActiveX control in your application** — The AppBuilder allows you to select the ActiveX control in design mode and place an instance of it into your application. It also allows you to set the values the properties of the control will initially assume at runtime, and saves these values in a separate binary file. This file (by default) has the same name as your application file with the `.wrx` extension. The `.wrx` file contains the definitions of all ActiveX control instances in the corresponding application (`.w`) file.

Aside from using the AppBuilder to create ActiveX control instances, you can also code OCX event procedures and control-frame event triggers with minimal effort using the AppBuilder. The AppBuilder event list includes both ActiveX control events and ABL control-frame widget events. The control-frame event names appear first, followed by the ActiveX control event names prefixed by OCX.

Run-mode requirements

An ActiveX control is always in run mode when you execute a OpenEdge application that includes it. No license is required for run-time access to an ActiveX control. Thus, to deploy and execute an application that contains ActiveX controls, you must provide at least the following files:

- The `.w` file generated by the AppBuilder for your application, or the compiled r-code file
- The `.wrx` file saved with your application
- The `.ocx` file for each ActiveX control contained in your application
- Other DLL support files and files containing data and bitmaps that come installed with the ActiveX control

The `.wrx` file contains most of the information required to use each ActiveX control instance at runtime, often including references to bitmaps and other external files.

Programming requirements

For more information on programming COM objects in ABL, see Using COM Objects in ABL on page 523. For information specific to Automation objects, see ActiveX Automation Support on page 543 and for information specific to ActiveX controls, see ActiveX Control Support on page 551.

Note: On non-Windows systems, any ABL code that references COM objects can compile, but generates run-time errors when executed. Therefore, isolate any COM object references in multi-platform code by using Preprocessor directives.

Sockets

Sockets are software communication end-points that allow one process to communicate with another on the same machine or across a network. ABL implementation supports TCP/IP sockets in ABL that allow one ABL application to establish a connection with another ABL or non-ABL application, and thus to communicate with that application on the same machine or across a network.

ABL socket implementation also includes support for Transport Layer Security (TLS) connections. This support includes attributes for specifying the TLS configuration for an ABL client or ABL server TLS session and utilities for managing key and certificate stores for OpenEdge TLS clients and servers. For more information on TLS and OpenEdge TLS support, see *Introduction to Security and Auditing*.

Note: TLS incurs heavy performance penalties, depending on the client, server, and network resources and load.

Reasons to use sockets

Sockets allow your ABL application to interact with other applications built using any language and deployed on any network machine using a standard communications model. ABL sockets are integrated with the ABL event model so you can use a single mechanism to handle user-interface events, application server asynchronous request completion events, and socket events. Applications that lend themselves to socket communications with ABL include:

- Ticker tape applications for financial markets
- Data acquisition for manufacturing and processes
- Web servers
- Mail servers
- Any other message-based applications

Alternative ABL mechanisms that allow socket access include the Host Language Call Interface (HLC) and shared library access described in this manual. However, ABL sockets provide a native ABL mechanism that is much easier to program and that is well-integrated with the ABL event model.

The ABL implementation provides low-level access to TCP/IP sockets. In ABL, a socket client and server each send and receive data as a stream of bytes, accessed as a `MEMPTR` data type. The formatting of this data stream is entirely application dependent. You can marshal and unmarshal data streams according to your application requirements using the ABL statements and functions available to manipulate `MEMPTR` data (see Using MEMPTR to reference external data on page 413).

Connection model

TCP/IP sockets use a connection model, where a client (*socket client*) seeks to establish a connection with a server (*socket server*). Once established, the socket client and server communicate over this connection in a peer-to-peer fashion by sending and receiving data streams over the established connection.

ABL socket clients

An ABL socket client can:

- Establish a connection with a server
- Write and read data on a connection
- Disconnect from the server

ABL TLS clients

In addition to the basic ABL socket client capabilities, an ABL TLS client can:

- Access an OpenEdge-supported public key certificate store to properly validate communications with a TLS server (ABL or non-ABL)
- Turn off verification of the TLS server (host) identity
- Specify that the same session ID cannot be re-used for each new connection on the same socket

ABL socket servers

An ABL socket server can:

- Notify ABL to listen and accept connections on a specified port
- Have ABL notify the server of new connections from clients
- Write and read data on a connection
- Disconnect from the client

ABL TLS servers

In addition to the basic ABL socket server capabilities, an ABL TLS server can:

- Identify and access a protected private key and digital certificate store that the server can use to assert its identity and to authorize access by TLS clients (ABL or non-ABL)
- Determine how TLS sessions with clients are managed, including such automatic features as the time-out interval for idol sessions

Server socket and socket objects

To enable ABL applications to access sockets, ABL supports two types of objects:

- **Server socket objects** — An ABL object that a socket server creates only to listen for client connection requests. An ABL socket server receives notification of client connections in the form of events.

  For more information, see "Server socket object handle" in the *ABL Reference*.

- **Socket objects** — An ABL object that represents a TCP/IP socket. TCP/IP sockets are the communication endpoints of a connection. Both socket clients and servers use socket objects to read and write data on a connection. In ABL, a socket application can detect the arrival of data on a socket in the form of events. (However, this is not required.)

  For more information, see "Socket object handle" in the *ABL Reference*.

Using socket objects

Using methods on the socket object, an ABL application can write and read any available data on a socket at any time, as long as the socket connection is active. Thus, the application can detect the arrival of data using events or not, depending on application requirements.

Both socket and server socket handles provide a variety of additional methods and attributes that you can use to monitor and control socket communications. For example, you can temporarily disable events on a socket object to run more efficiently when a client or server does not need to receive data on the connection. You can also temporarily or permanently disable events on a server socket when you no longer want the server to respond to client connection requests. Other methods and attributes allow you to set and monitor additional socket communications options and conditions.

ABL socket event model

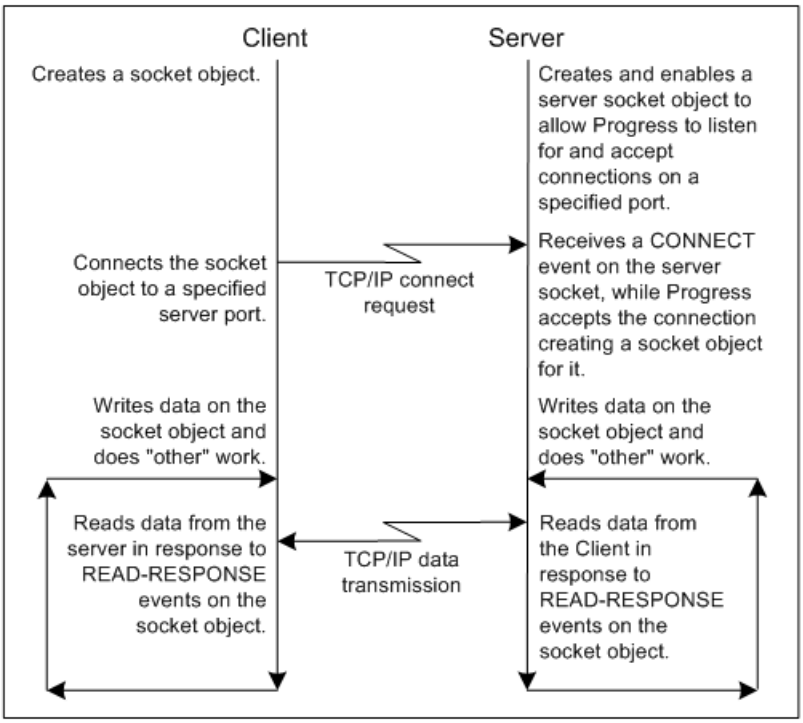
The ABL socket event model includes two types of ABL events:

- `CONNECT event`—Posted on a server socket object when a client seeks to establish a socket connection. As part of generating this event, ABL creates a socket object for the server to communicate with the client on the new connection.
- `READ-RESPONSE event`—Posted on a socket object when data is available to be read on the socket.

Using events to connect and communicate

Like all ABL events, these CONNECT and READ-RESPONSE events are handled in the context of an input-blocking or PROCESS EVENTS statement. The following figure shows how two ABL applications, one acting as a socket server and the other as a socket client, can use these events to communicate across a single connection.

Figure 39: ABL socket event model



The server creates and enables a server socket object for listening on a specified port. The client creates a socket object and attempts to connect that socket to the server port used by the server socket. The server accepts the connection request and runs a specified CONNECT event handler in response to the CONNECT event. Once a connection is established, both the client and server can read and write data to each other using their connected socket objects. Then the server socket object resumes listening for connection requests from other clients.

The socket object that the client creates and the socket object created on the server in response to the CONNECT event both reference the same TCP/IP connection. Using the ABL event model, the client and server can each receive notifications of data from the other within a READ-RESPONSE event handler that runs in response to a READ-RESPONSE event on the socket object.

Programming requirements

To use sockets in ABL:

- Socket clients and servers must run locally or remotely in a TCP/IP network environment.
- A potential server must have a TCP/IP port available on which to listen for connections.
- A socket server can have only one server socket object enabled to listen for connections.
- Application server and WebSpeed procedures cannot function as socket servers—they can only function as socket clients.
- A potential client must:

- Know the hostname or IP address of the machine where the server it wants to connect to is running.
- Know the TCP/IP port where the server is listening for connections.
- For TLS sessions, have access to a public key certificate store as maintained by the OpenEdge features for managing certificate stores. For more information, see *Introduction to Security and Auditing*.

For more information on programming with sockets, see Sockets on page 579.

11

System Clipboard

The *system clipboard* is a feature provided on most window systems that allows the user to transfer data between one widget (or application) and another using standard mouse and keyboard operations. Each application typically provides some form of program support for how these operations interact with it. You can provide this support in a OpenEdge application using the `CLIPBOARD` system handle.

ABL (Advanced Business Language) supports clipboard operations between OpenEdge and other applications in Windows. In character mode, ABL supports clipboard operations within a OpenEdge application. For more information on how clipboard operations work in Windows, see the *Microsoft Windows User's Guide*.

For details, see the following topics:

- CLIPBOARD system handle
- Single-item data transfers
- Multiple-item data transfers

CLIPBOARD system handle

The `CLIPBOARD` system handle allows you to transfer data between the window system clipboard and your OpenEdge application. Using the `CLIPBOARD` attributes, you can paste (read) data from the system clipboard to an ABL field or variable, and copy or cut (write) data from a field or variable to the clipboard. These cut, copy, and paste data transfers are the basic *clipboard operations* typically provided by the system clipboard to the user.

In ABL, you also have a choice of two data transfer modes to implement these operations—single-item transfers and multiple-item transfers. In *single-item transfers*, a single write to the clipboard immediately replaces all data in the clipboard, and a single read from the clipboard returns all data in the clipboard to the OpenEdge application. In *multiple-item transfers*, you can format the data transfer into multiple rows of multiple items. Each write to the clipboard adds an item to a tab- and newline-separated list of clipboard items; each read from the clipboard returns one tab- or newline-separated item to your OpenEdge application. This mode is especially useful to allow users to transfer aggregate units of data, in one step, between OpenEdge and other applications (such as spreadsheets) that also support aggregate clipboard operations in a similar way.

These data transfers are accomplished with the help of the CLIPBOARD handle attributes listed in the following table.

Table 49: CLIPBOARD handle attributes

Attribute	Type	Readable	Setable
AVAILABLE-FORMATS	CHARACTER	•	–
ITEMS-PER-ROW	INTEGER	•	•
MULTIPLE	LOGICAL	•	•
NUM-FORMATS	INTEGER	•	–
TYPE	CHARACTER	•	–
VALUE	CHARACTER	•	•

The following sections provide an overview of the CLIPBOARD handle attributes. For a complete description of the CLIPBOARD handle and its attributes, see the CLIPBOARD Handle reference entry in *ABL Reference*.

AVAILABLE-FORMATS Attribute

The AVAILABLE attribute returns a comma-separated list of the available formats for the data stored in the clipboard. The supported formats include:

- **PRO_TEXT** — Specifies the standard text format on your system (CF_TEXT for Windows)
- **PRO_MULTIPLE** — Specifies that the data in the clipboard contains tab or newline characters, and thus can be read as multiple items

ITEMS-PER-ROW Attribute

The ITEMS-PER-ROW attribute specifies how many tab-separated items are formatted in a newline-separated row for multiple-item writes to the clipboard. This attribute has no effect on multiple-item reads from the clipboard. The user or program that originally moves the data to the clipboard must format the data according to how your application expects to read it.

MULTIPLE Attribute

Setting the MULTIPLE attribute to TRUE starts a multiple-item transfer. Setting it to FALSE ends the multiple-item transfer and readies the clipboard for single-item transfers. After multiple writes, a FALSE setting transfers the formatted list of items to the clipboard and resets the ITEMS-PER-ROW attribute to 1. After multiple reads, a FALSE setting allows you to restart reading from the first item after resetting the attribute to TRUE.

NUM-FORMATS Attribute

The NUM-FORMATS attribute returns the number of data formats available for reading data from the clipboard. If there is no data in the clipboard, the value is 0.

TYPE Attribute

The TYPE attribute returns the widget type of the CLIPBOARD handle, which is the standard type for system handles, "PSEUDO-WIDGET". This attribute has no effect on data transfers, and is essentially used for documentation.

VALUE attribute

The VALUE attribute provides access to the system clipboard data. Set the attribute to the value of a field or variable you want to cut or copy to the clipboard. To cut, set the source data item to the Unknown value (?) or the null string ("") after you set the attribute. Assign the value of the attribute to a field or variable to which you want to paste data from the clipboard. If there is no data in the clipboard or you read the last item in a multiple read, the attribute returns the Unknown value (?) to the data item.

A single-item write to this attribute immediately replaces all previous data in the clipboard. A multiple-item write to this attribute appends the data item to a buffered list of items. Once you set the MULTIPLE attribute to FALSE, the CLIPBOARD handle formats the list according to the value of ITEMS-PER-ROW and replaces all previous clipboard data with it. Both single-item and multiple-item reads are nondestructive to data in the clipboard.

Single-item data transfers

Each single-item data transfer moves data between the ABL data item and the clipboard. During a paste operation, all data stored in the clipboard is transferred to the data item. During a cut or copy operation, the value of the ABL data item replaces any and all data in the clipboard.

Clipboard operations are typically invoked by **Cut**, **Copy**, and **Paste** options on an **Edit** menu. You can implement these operations in a general way using the FOCUS system handle (for more information, see the FOCUS Handle reference entry in *ABL Reference*). This allows you to program two types of actions that:

- Determine what clipboard operations are available (enabled and disabled) at any point
- Specify how each clipboard operation is implemented when it is available

Enabling and disabling clipboard operations

You can enable and disable clipboard operations based on the type of widget that currently has the input focus (FOCUS:TYPE attribute). For example, you might disable the pasting (inserting) of values into a selection list, but enable the copying of selected items from the list. You typically configure your clipboard operations in the trigger block of the MENU-DROP event for your **Edit** menu. This ensures that you enable or disable menu options based on the latest selection action that the user has committed in the current input widget (for example, selected a radio set or a range of text in an editor widget).

The following code example suggests a possible scenario for enabling and disabling cut, copy, and paste operations. It defines an **Edit** menu (EditMenu) with **Cut**, **Copy**, and **Paste** options assigned to the corresponding menu items EM_Cut, EM_Copy, and EM_Paste. When the user opens the **Edit** menu (ON MENU-DROP OF MENU EditMenu), the code determines the available options from the state of the field-level widget that has the current input focus.

For example, if an editor widget has the input focus (FOCUS:TYPE = "EDITOR"), then the Cut and Copy options are available only if the user has text selected within the widget (LENGTH(FOCUS:SELECTION-TEXT) > 0). If a radio set, selection list, slider, or toggle box has the input focus, then only the Copy option is enabled. You could make the Cut and Paste options meaningful for radio sets and selection lists, for example, by recreating dynamic radio sets or removing and inserting items in selection lists.

Although these topics provide useful examples, your code to determine the available clipboard operations can vary widely depending on your application. For example:

```
DEFINE MENU EditMenu
  MENU-ITEM EM_Cut   LABEL "&Cut "
  MENU-ITEM EM_Copy  LABEL "C&opy "
  MENU-ITEM EM_Paste LABEL "&Paste ".

ON MENU-DROP OF MENU EditMenu DO:
  IF FOCUS:TYPE = "EDITOR" THEN DO:
    MENU-ITEM EM_Cut:SENSITIVE IN MENU EditMenu =
      IF  LENGTH(FOCUS:SELECTION-TEXT) > 0 THEN TRUE ELSE FALSE.
    MENU-ITEM Em_Copy:SENSITIVE IN MENU EditMenu =
      IF  LENGTH(FOCUS:SELECTION-TEXT) > 0 THEN TRUE ELSE FALSE.
    MENU-ITEM EM_Paste:SENSITIVE IN MENU EditMenu =
      IF CLIPBOARD:NUM-FORMATS > 0 THEN TRUE ELSE FALSE.
  END.
  ELSE IF FOCUS:TYPE = "RADIO-SET" OR
          FOCUS:TYPE = "SELECTION-LIST" OR
          FOCUS:TYPE = "SLIDER" OR
          FOCUS:TYPE = "TOGGLE-BOX" THEN DO:
    MENU-ITEM EM_Cut:SENSITIVE IN MENU EditMenu   = FALSE.
    MENU-ITEM Em_Copy:SENSITIVE IN MENU EditMenu  = TRUE.
    MENU-ITEM Em_Paste:SENSITIVE IN MENU EditMenu = FALSE.
  END.
  ELSE IF FOCUS:TYPE = "FILL-IN" THEN DO:
    MENU-ITEM EM_Cut:SENSITIVE IN MENU EditMenu =
      IF LENGTH(FOCUS:SCREEN-VALUE) > 0 THEN TRUE ELSE FALSE.
    MENU-ITEM Em_Copy:SENSITIVE IN MENU EditMenu =
      IF LENGTH(FOCUS:SCREEN-VALUE) > 0 THEN TRUE ELSE FALSE.
    MENU-ITEM EM_Paste:SENSITIVE IN MENU EDitMenu =
      IF CLIPBOARD:NUM-FORMATS > 0 THEN TRUE ELSE FALSE.
  END.
  ELSE DO:    MENU-ITEM EM_Cut:SENSITIVE IN MENU EditMenu   = FALSE.
    MENU-ITEM EM_Copy:SENSITIVE IN MENU EditMenu  = FALSE.
    MENU-ITEM EM_Paste:SENSITIVE IN MENU EditMenu = FALSE.
  END.
END. /* ON MENU-DROP IN EditMenu */
```

Implementing single-item transfers

You can implement each clipboard operation based on the type of widget that currently has the input focus (FOCUS:TYPE) and the state of its text selection and other attributes. For example, you can decide whether to copy all or part of an editor widget value by the values of the SELECTION-START and SELECTION-END attributes. You typically implement each clipboard operation in the trigger block of the CHOOSE event for the corresponding **Edit** menu option. In this way, the user can only perform a clipboard operation associated with an **Edit** menu option that is enabled.

The following code example implements the clipboard operations enabled by the code example in the previous section. Note that for editor widgets (FOCUS:TYPE = "EDITOR"), if the user has text selected, the procedure transfers data between the clipboard and the SELECTION-TEXT rather than the VALUE attribute itself.

Paste operations

For example, the paste operation (ON CHOOSE OF MENU-ITEM EM_Paste) replaces only the selected text (rather than the whole text) in an editor widget with the data in the clipboard. (In a fill-in widget, paste operations always replace all data in the widget.)

Cut operations

For cut operations (ON CHOOSE OF MENU-ITEM EM_Cut), the procedure sets the appropriate widget attribute (SELECTION-TEXT or VALUE) to the empty string after transferring the data to the clipboard. The corresponding text disappears from the display as the Cut operation completes.

Copy operations

Copy operations (ON CHOOSE OF MENU-ITEM EM_Copy) are similar to cut operations except that they leave the FOCUS data unchanged. However, if the data to be copied is a radio set, the example assumes that the character value of the radio set label visible on the display (FOCUS:LABEL) is what the user wants to copy rather than its value (FOCUS:VALUE). This is a useful implementation where the radio set represents an integer and the FOCUS:VALUE attribute contains a right-justified integer string. For example:

```
DEFINE VARIABLE lStat AS LOGICAL NO-UNDO.
DEFINE MENU EditMenu
  MENU-ITEM EM_Cut   LABEL "&Cut "
  MENU-ITEM EM_Copy  LABEL "C&opy "
  MENU-ITEM EM_Paste LABEL "&Paste ".

ON CHOOSE OF MENU-ITEM EM_Cut IN MENU EditMenu DO:
  IF FOCUS:TYPE = "EDITOR" THEN DO:
    IF FOCUS:SELECTION-START <> FOCUS:SELECTION-END THEN DO:
      CLIPBOARD:VALUE = FOCUS:SELECTION-TEXT.
      lStat           = FOCUS:REPLACE-SELECTION-TEXT("").
    END.
    ELSE DO:
      CLIPBOARD:VALUE    = FOCUS:SCREEN-VALUE.
      FOCUS:SCREEN-VALUE = "".
    END.
  END.
  ELSE DO: /* For FILL-IN */
    CLIPBOARD:VALUE    = FOCUS:SCREEN-VALUE.
    FOCUS:SCREEN-VALUE = "".
  END.
END. /* ON CHOOSE OF MENU-ITEM EM_Cut */

ON CHOOSE OF MENU-ITEM EM_Copy IN MENU EditMenu DO:
  IF FOCUS:TYPE = "EDITOR" THEN
    IF FOCUS:SELECTION-START <> FOCUS:SELECTION-END THEN
      CLIPBOARD:VALUE = FOCUS:SELECTION-TEXT.
    ELSE
      CLIPBOARD:VALUE = FOCUS:SCREEN-VALUE.
  ELSE IF FOCUS:TYPE = "RADIO-SET" THEN
    CLIPBOARD:VALUE = ENTRY(LOOKUP(FOCUS:SCREEN-VALUE,
      FOCUS:RADIO-BUTTONS) - 1, FOCUS:RADIO-BUTTONS).
  ELSE IF FOCUS:TYPE = "TOGGLE-BOX" THEN
    IF FOCUS:SCREEN-VALUE = "yes" THEN
      CLIPBOARD:VALUE = FOCUS:LABEL + " selected.".
    ELSE
      CLIPBOARD:VALUE = FOCUS:LABEL + " not selected.".
  ELSE /* For FILL-IN */
    CLIPBOARD:VALUE = FOCUS:SCREEN-VALUE.
END. /* ON CHOOSE OF MENU-ITEM EM_Copy */
ON CHOOSE OF MENU-ITEM EM_Paste IN MENU EditMenu DO:
  IF FOCUS:TYPE = "EDITOR" THEN DO:
    IF FOCUS:SELECTION-START <> FOCUS:SELECTION-END THEN
      lStat = FOCUS:REPLACE-SELECTION-TEXT(CLIPBOARD:VALUE).
    ELSE
      lStat = FOCUS:INSERT-STRING(CLIPBOARD:VALUE).
  END.
  ELSE /* For FILL-IN */
    FOCUS:SCREEN-VALUE = CLIPBOARD:VALUE.
END. /* ON CHOOSE OF MENU-ITEM EM_Paste */
```

Single-item transfer example

The `i-clpbrd.p` procedure uses the clipboard operation implementation described in the previous section. It both demonstrates the capabilities of that design and serves as a primer for other design alternatives. (To run `i-clpbrd.p` in character mode, comment out all statements that reference MainWindow and assign the `MENU MainMenu:HANDLE` attribute to the `CURRENT-WINDOW:MENUBAR` attribute.)

i-clpbrd.p

```
DEFINE VARIABLE lStat      AS LOGICAL   NO-UNDO.
DEFINE VARIABLE MainWindow AS HANDLE    NO-UNDO.
DEFINE VARIABLE MyEditor   AS CHARACTER NO-UNDO
  VIEW-AS EDITOR SIZE 20 BY 4 SCROLLBAR-VERTICAL.
DEFINE VARIABLE Fillin     AS CHARACTER NO-UNDO FORMAT "x(20)".
DEFINE VARIABLE TogDemo    AS LOGICAL   NO-UNDO EXTENT 2
  INITIAL ["FALSE", "TRUE"]
  LABEL "Pick-Me1", "Pick-Me2"
  VIEW-AS TOGGLE-BOX.
DEFINE VARIABLE Radios     AS INTEGER  NO-UNDO INITIAL 3
  LABEL "Time-O-Day"
  VIEW-AS RADIO-SET RADIO-BUTTONS "1pm", 1, "2pm", 2, "3pm", 3.
DEFINE VARIABLE Slider1    AS INTEGER  NO-UNDO
  VIEW-AS SLIDER SIZE-PIXELS 120 by 40
  MAX-VALUE 100 MIN-VALUE 10 LABEL "Slide Me:".
DEFINE VARIABLE Slider2    AS INTEGER  NO-UNDO
  VIEW-AS SLIDER SIZE-PIXELS 120 by 80
  MAX-VALUE 1000 MIN-VALUE 100 VERTICAL LABEL "Slide Me:".
DEFINE VARIABLE SelectList AS CHARACTER NO-UNDO
  VIEW-AS SELECTION-LIST SINGLE SIZE 23 BY 7
  LIST-ITEMS "Line 1", "Line 2", "Line 3", "Line 4", "Line 5", "Line 6",
    "Line 7", "Line 8", "Line 9", "Line 10", "Line 11", "Line 12",
    "Line 13", "Line 14", "Line 15".

DEFINE SUB-MENU FileMenu
  MENU-ITEM FM_New       LABEL "&New"
  MENU-ITEM FM_Open      LABEL "&Open... "
  RULE
  MENU-ITEM FM_Save      LABEL "&Save "
  MENU-ITEM FM_Save_as   LABEL "Save &As... "
  RULE
  MENU-ITEM FM_Exit      LABEL "E&xit ".

DEFINE SUB-MENU EditMenu
  MENU-ITEM EM_Cut   LABEL "&Cut "
  MENU-ITEM EM_Copy  LABEL "C&opy "
  MENU-ITEM EM_Paste LABEL "&Paste ".
DEFINE MENU MainMenu
  MENUBAR
  SUB-MENU  FileMenu LABEL "&File "
  SUB-MENU  EditMenu LABEL "&Edit ".

DEFINE BUTTON b_OK     LABEL "   OK   ".
DEFINE BUTTON b_Cancel LABEL " CANCEL ".

FORM
  "Enter Text Here" AT ROW 1 COLUMN 2
  MyEditor AT ROW 2 COLUMN 2
  "Fill In Here" AT ROW 1 COLUMN 39
  Fillin AT ROW 2 COLUMN 39
  TogDemo[1] AT ROW 2 COLUMN 25
  TogDemo[2] AT ROW 4 COLUMN 25
  Radios AT ROW 7 COLUMN 2
  "Selection List" AT ROW 6 COLUMN 22
  SelectList AT ROW 7 COLUMN 17
  Slider2 AT ROW 7 COLUMN 45
  Slider1 AT ROW 12 COLUMN 43
```

```
  b_OK AT ROW 12 COLUMN 2
  b_Cancel AT ROW 14 COLUMN 2 SKIP 0.5)
  WITH FRAME MainFrame NO-LABEL CENTERED WIDTH-CHARS 80.

ON CHOOSE OF b_OK IN FRAME MainFrame MESSAGE "OK pressed".
ON CHOOSE OF b_Cancel IN FRAME MainFrame MESSAGE "CANCEL pressed".
ON CHOOSE OF MENU-ITEM FM_Exit IN MENU FileMenu STOP.

/****** Begin Clipboard Code ******/
ON MENU-DROP OF MENU EditMenu DO:
  IF FOCUS:TYPE = "EDITOR" THEN DO:
    MENU-ITEM EM_Cut:SENSITIVE IN MENU EditMenu =
      IF LENGTH(FOCUS:SELECTION-TEXT) > 0 THEN TRUE ELSE FALSE.
    MENU-ITEM Em_Copy:SENSITIVE IN MENU EditMenu =
      IF LENGTH(FOCUS:SELECTION-TEXT) > 0 THEN TRUE ELSE FALSE.
    MENU-ITEM EM_Paste:SENSITIVE IN MENU EditMenu =
      IF CLIPBOARD:NUM-FORMATS > 0 THEN TRUE ELSE FALSE.
  END.
  ELSE IF FOCUS:TYPE = "RADIO-SET" OR
          FOCUS:TYPE = "SELECTION-LIST" OR
          FOCUS:TYPE = "SLIDER" OR
          FOCUS:TYPE = "TOGGLE-BOX" THEN DO:
    MENU-ITEM EM_Cut:SENSITIVE IN MENU EditMenu   = FALSE.
    MENU-ITEM Em_Copy:SENSITIVE IN MENU EditMenu  = TRUE.
    MENU-ITEM Em_Paste:SENSITIVE IN MENU EditMenu = FALSE.
  END.

  ELSE IF FOCUS:TYPE = "FILL-IN" THEN DO:
    MENU-ITEM EM_Cut:SENSITIVE IN MENU EditMenu =
      IF LENGTH(FOCUS:SCREEN-VALUE) > 0 THEN TRUE ELSE FALSE.
    MENU-ITEM Em_Copy:SENSITIVE IN MENU EditMenu =
      IF LENGTH(FOCUS:SCREEN-VALUE) > 0 THEN TRUE ELSE FALSE.
    MENU-ITEM EM_Paste:SENSITIVE IN MENU EDitMenu =
      IF CLIPBOARD:NUM-FORMATS > 0 THEN TRUE ELSE FALSE.
  END.
  ELSE DO:
    MENU-ITEM EM_Cut:SENSITIVE IN MENU EditMenu = FALSE.
    MENU-ITEM EM_Copy:SENSITIVE IN MENU EditMenu = FALSE.
    MENU-ITEM EM_Paste:SENSITIVE IN MENU EditMenu = FALSE.
  END.
END. /* ON MENU-DROP IN EditMenu */

ON CHOOSE OF MENU-ITEM EM_Cut IN MENU EditMenu DO:
  IF FOCUS:TYPE = "EDITOR" THEN DO:
    IF FOCUS:SELECTION-START <> FOCUS:SELECTION-END THEN DO:
      CLIPBOARD:VALUE = FOCUS:SELECTION-TEXT.
      lStat           = FOCUS:REPLACE-SELECTION-TEXT("").
    END.
    ELSE DO:
      CLIPBOARD:VALUE    = FOCUS:SCREEN-VALUE.
      FOCUS:SCREEN-VALUE = "".
    END.
  END.
  ELSE DO: /* For FILL-IN */
    CLIPBOARD:VALUE    = FOCUS:SCREEN-VALUE.
    FOCUS:SCREEN-VALUE = "".
  END.
END. /* ON CHOOSE OF MENU-ITEM EM_Cut */

ON CHOOSE OF MENU-ITEM EM_Copy IN MENU EditMenu DO:
  IF FOCUS:TYPE = "EDITOR" THEN
    IF FOCUS:SELECTION-START <> FOCUS:SELECTION-END THEN
      CLIPBOARD:VALUE = FOCUS:SELECTION-TEXT.
    ELSE
      CLIPBOARD:VALUE = FOCUS:SCREEN-VALUE.
  ELSE IF FOCUS:TYPE = "RADIO-SET" THEN
    CLIPBOARD:VALUE = ENTRY(LOOKUP(FOCUS:SCREEN-VALUE,
      FOCUS:RADIO-BUTTONS) - 1, FOCUS:RADIO-BUTTONS).
  ELSE IF FOCUS:TYPE = "TOGGLE-BOX" THEN
```

```
    IF FOCUS:SCREEN-VALUE = "yes" THEN
      CLIPBOARD:VALUE = FOCUS:LABEL + " selected.".
    ELSE
      CLIPBOARD:VALUE = FOCUS:LABEL + " not selected.".
  ELSE /* For FILL-IN */
    CLIPBOARD:VALUE = FOCUS:SCREEN-VALUE.
END. /* ON CHOOSE OF MENU-ITEM EM_Copy */

ON CHOOSE OF MENU-ITEM EM_Paste IN MENU EditMenu DO:
  IF FOCUS:TYPE = "EDITOR" THEN DO:
    IF FOCUS:SELECTION-START <> FOCUS:SELECTION-END THEN
      lStat = FOCUS:REPLACE-SELECTION-TEXT(CLIPBOARD:VALUE).
    ELSE
      lStat = FOCUS:INSERT-STRING(CLIPBOARD:VALUE).
  END.
  ELSE /* For FILL-IN */
    FOCUS:SCREEN-VALUE = CLIPBOARD:VALUE.
END. /* ON CHOOSE OF MENU-ITEM EM_Paste */
/****** End Clipboard Code ******/

CREATE WINDOW MainWindow
  ASSIGN
    X = 0
    Y = 0
    MENUBAR = MENU MainMenu:HANDLE
    TITLE = "CLIPBOARD SUPPORT".

CURRENT-WINDOW = MainWindow.

ON WINDOW-CLOSE OF MainWindow STOP.

ENABLE ALL WITH FRAME MainFrame.
STATUS DEFAULT "Widgets and the Clipboard".

WAIT-FOR CHOOSE OF b_Cancel IN FRAME MainFrame.
DELETE WIDGET MainWindow.
```

Multiple-item data transfers

Each multiple-item data transfer moves data between one or more ABL data items and the clipboard. During a paste operation, all data stored in the clipboard is transferred to the data items. During a cut or copy operation, the values of the selected ABL data items replace any and all data in the clipboard.

Multiple-item clipboard operations are typically invoked by **Cut**, **Copy**, and **Paste** options on an **Edit** menu, just like single-item operations. There are two basic techniques you can use to implement multiple transfers in a procedure:

- **Widget-based transfers** — Provide a selection mode that allows the user to select eligible widgets for the selected clipboard operation. After the user selects and confirms the widgets that are valid for the operation, the operation proceeds.
- **Data-based transfers** — Provide options to transfer data directly between the OpenEdge database and the clipboard. This is the most common type of multiple-item data transfer.

To implement any multiple-item data transfer:

1. Determine the clipboard operation to perform and the data items to participate in the operation.
2. Set the `CLIPBOARD` handle `MULTIPLE` attribute to `TRUE`. For Cut/Copy (write) operations, set the `ITEMS-PER-ROW` attribute to the number of items in each line of data written to the clipboard.

3. For each data item, assign the appropriate data item value (screen or record buffer) to the VALUE attribute for Cut/Copy operations or assign the VALUE attribute to the data item for Paste (read) operations.
4. Set the MULTIPLE attribute to FALSE to complete the operation. This resets the item pointer to the beginning of the clipboard for a Cut/Copy operation and writes the item-formatted data to the clipboard for a Paste operation. (This also resets the ITEMS-PER-ROW attribute.)

Widget-based transfers

The techniques for implementing widget-based multiple-item transfers are very similar to those used for single-item transfers (see Single-item data transfers on page 439). The basic difference is in the order of operations and the extra steps to provide widget selection and confirmation before the selected operation proceeds to completion. You might implement a widget-based data transfer according to the following processing model:

1. The user chooses the **Cut**, **Copy**, or **Paste** option from a multiple-item transfer menu.
2. On the CHOOSE event for the chosen transfer option, the procedure enables all eligible widgets for selection (SELECTABLE attribute = TRUE) and makes all other nonparticipating widgets insensitive (SENSITIVE attribute = FALSE).
3. The user can now only select widgets for the selected data transfer option and confirm the selection.
4. When the user is finished selecting widgets, they invoke an option (for example, a button or menu item) that confirms and allows the operation to proceed to completion. (The user might also invoke an option to cancel the current operation and return to other application functions.)
5. On confirmation of widget selection, the procedure:
   a. Sets the appropriate CLIPBOARD handle attributes for the selected operation. This requires setting the MULTIPLE attribute to TRUE, and for Cut or Copy operations (write transfers) setting the ITEMS-PER-ROW attribute to format the data into lines of tab-separated items.
   b. Iterates through the widget list assigning each selected widget's SCREEN-VALUE attribute to the CLIPBOARD handle VALUE attribute for a Cut/Copy operation, or assigning the VALUE attribute to each SCREEN-VALUE attribute for a Paste operation. (You can implement the iteration through the widget list using either the NEXT/PREVIOUS-SIBLING or NEXT/PREVIOUS-TAB-ITEM attribute to return and save the handle of each succeeding widget in the list.)
   c. Completes the operation by setting the MULTIPLE attribute to FALSE and disabling selection and enabling sensitivity for all widgets (SELECTABLE = FALSE and SENSITIVE = TRUE).
6. The user can now perform other application functions.

In this implementation, the essential tasks in the multiple-item transfer are included in Item 5. Of course, there are many variations of this process that you can implement, such as providing preselected widget lists from which the user can choose (eliminating the need for the selection mode enabled in Item 2). This latter approach can employ a processing model similar to that used for data-based transfers.

Data-based transfers

A typical data-based transfer differs from a widget-based transfer in that you directly reference the fields in a database (the record buffer) rather than the widgets visible on the display (the screen buffer). You might implement this type of transfer according to the following processing model:

1. The user chooses tables and fields from the database to participate in the selected operation.
2. The user indicates the operation (Cut, Copy, or Paste) to perform.

3. The procedure sets the CLIPBOARD handle MULTIPLE attribute to TRUE, and for Cut or Copy (write) operations, sets the ITEMS-PER-ROW attribute to the number of fields in each record participating in the transfer.
4. For a Cut/Copy operation, the procedure assigns each field to the VALUE attribute for each (selected) record in the table. For a Paste operation, the procedure assigns the VALUE attribute to the fields in each record created or updated in the table.
5. The procedure sets the MULTIPLE attribute to FALSE, ending the operation.

Note that for Cut/Copy operations on database fields you must access any noncharacter fields using the STRING function. For Paste (read) operations, your procedure must depend on the user to provide appropriately formatted data items in the clipboard. Your database validation functions can help to catch and respond to bad data in the clipboard.

Multiple-item transfer example

The i-clpmui.p procedure demonstrates the essential elements of a multiple-item data transfer. It implements a basic data-based Copy operation using the customer table of the sports2000 database. You can test the result by running the procedure and pasting the result into a window system editor like Notepad or Wordpad in Windows.

i-clpmul.p

```
CLIPBOARD:MULTIPLE      = TRUE.
CLIPBOARD:ITEMS-PER-ROW = 11.

FOR EACH Customer NO-LOCK:
  CLIPBOARD:VALUE = STRING(Customer.CustNum).
  CLIPBOARD:VALUE = Customer.Name.
  CLIPBOARD:VALUE = Customer.Address.
  CLIPBOARD:VALUE = Customer.Address2.
  CLIPBOARD:VALUE = Customer.City.
  CLIPBOARD:VALUE = Customer.State.
  CLIPBOARD:VALUE = Customer.PostalCode.
  CLIPBOARD:VALUE = Customer.Country.
  CLIPBOARD:VALUE = Customer.Phone.
  CLIPBOARD:VALUE = STRING(Customer.Balance).
  CLIPBOARD:VALUE = STRING(Customer.CreditLimit).
END.

CLIPBOARD:MULTIPLE = FALSE.
```

12

Providing Help for OpenEdge Applications

Online help provides users with immediate access to information as they work with a software application. OpenEdge provides multiple ways to supply online help to applications.

These topics explain how you can implement:

- **Field-level help** — Short strings of text that describe the functions of the field-level objects in your application's user interface. Field-level help includes status area messages and ToolTips.
- **A complete online help system** — Help files that can provide large amounts of information about your entire application.

You can use both of these help delivery mechanisms or just one, depending on the kind of information you want to provide your users.

For details, see the following topics:

- Field-level online help
- Online help systems

Field-level online help

OpenEdge supports two techniques for implementing field-level help for the user interface of your application: status messages and ToolTips.

Status messages

A status message is a string of text that appears in the status area of a window and describes the function of the field-level object that has input focus.

Note:

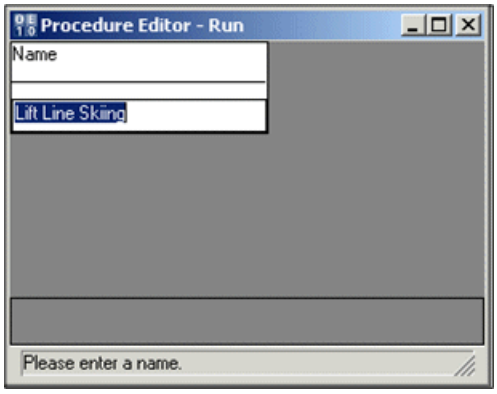
The status area is an optional feature of an OpenEdge application window. It displays one line of message text at the bottom of the window. Its appearance is controlled by the STATUS-AREA attribute. Thus, if the STATUS-AREA is disabled on a window, no help message appears. While a dialog box does not have a status area, help strings defined for fields on a dialog box are displayed in the status area of its parent window.

Also, when you use the HELP attribute to display help text for a widget, ABL (Advanced Business Language) overwrites any status text (defined with the STATUS statement) with the HELP text.

Status messages are a good way to provide cursory information about a database field or a field-level object in your application.

OpenEdge provides the ability to associate status messages with database fields and those field-level objects that can receive input focus. These messages are easy to implement and produce a preliminary level of help for your application. When the user tabs through certain field-level objects (fill-in fields, buttons, combo boxes, selection lists, editors, toggle boxes, radio sets, and sliders) **that are enabled for input** on an application interface, OpenEdge displays help text in the status area of the current window, as shown in the following figure.

Figure 40: Status bar message for a field-level object



There are three methods for creating help text in the status area:

- Defining help text for database fields in the schema definition via the Data Dictionary
- Creating help text for field-level widgets with ABL statements and the HELP attribute
- Creating help text for field-level widgets in the **Advanced Properties** dialog box in the AppBuilder

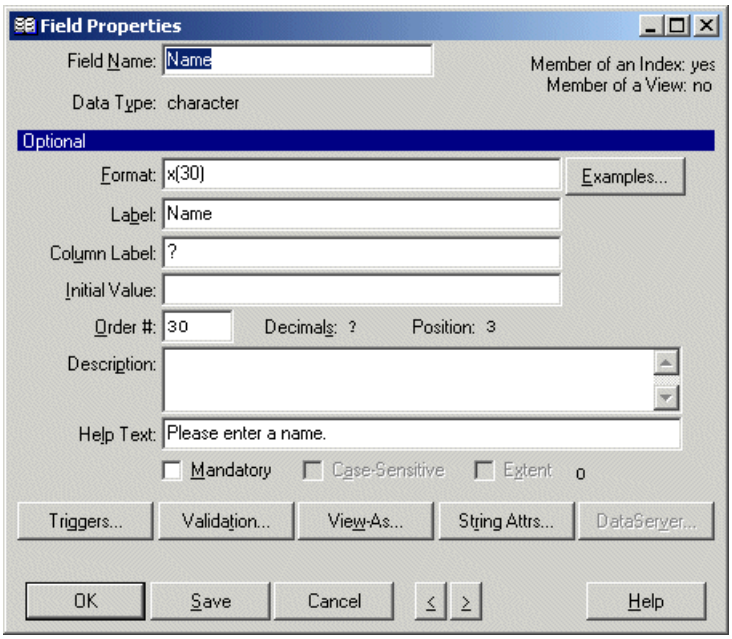
The sections that follow discuss these methods for creating help text.

Associating help text with database fields

By default, a data representation object associated with a database field displays the help text defined for that database field. You can define and store a default help string as part of the schema definition of a database field using the Data Dictionary. Creating help strings in the database schema provides a centralized location for help strings and makes them easy to maintain.

To define help text in a database field's schema definition:

1. Connect to your application database, for example, `sports2000`.
2. In the Data Dictionary, select a database table (for example, **Customer**) then click **Fields**.
3. Double-click a database field, for example, **Name**. The **Field Properties** dialog box appears:



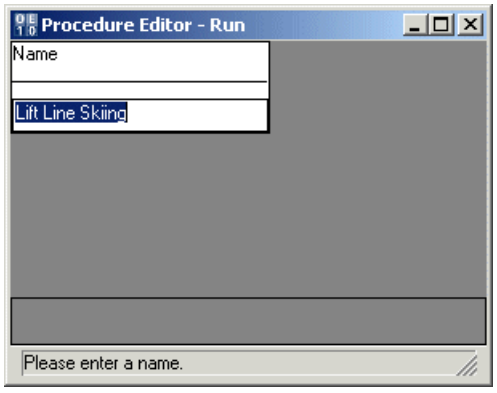
4. In the **Help Text** field, type the text string you want to associate with the database field, then click **OK**.

You can then run a test program to view the message in the status area. For example, you can open the Procedure Editor and run the following program:

```
FOR EACH Customer:
  UPDATE Customer.Name WITH 1 DOWN.
END.
```

You can then run this program to see your help text shown in the status area of the running window, as shown in the following figure.

Figure 41: Sample database field help text in the status area of a window



Note: Make sure that the test program involves an update to the database field. A program that just displays the field does not show the help message in the status area.

Specifying help text with ABL

The alternative to associating help strings with database fields is to define help strings as a part of field-level objects that contain the data. To attach a help string to a field-level object, use the HELP option of the Format phrase specified with the following ABL statements: DEFINE BROWSE, DEFINE TEMP-TABLE, DEFINE FRAME, ENABLE, FORM, PROMPT-FOR, SET, or UPDATE. The following code shows an example using the DEFINE FRAME statement:

```
DEFINE FRAME FRAME-A
   BUTTON-1 AT ROW 4 COL 12 HELP "Choose to cancel the operation and exit."
   WITH 1 DOWN NO-BOX OVERLAY SIDE-LABELS AT COL 1 ROW 1 SIZE 74 BY 11.
```

You can also use the HELP attribute to define help strings for field-level objects. For example, after the DEFINE FRAME statement, you can use the following code to change the help text for the button:

```
BUTTON-1:HELP = "This button cancels the operation and exits the window."
```

In ABL, the Format phrase HELP option and the HELP attribute let you define help strings for widgets not associated with database fields. Help strings defined with these options override any help strings specified for associated database fields in the Data Dictionary. For more information, see the Format Phrase and the HELP attribute reference entries in *ABL Reference*.

Defining help text with the AppBuilder

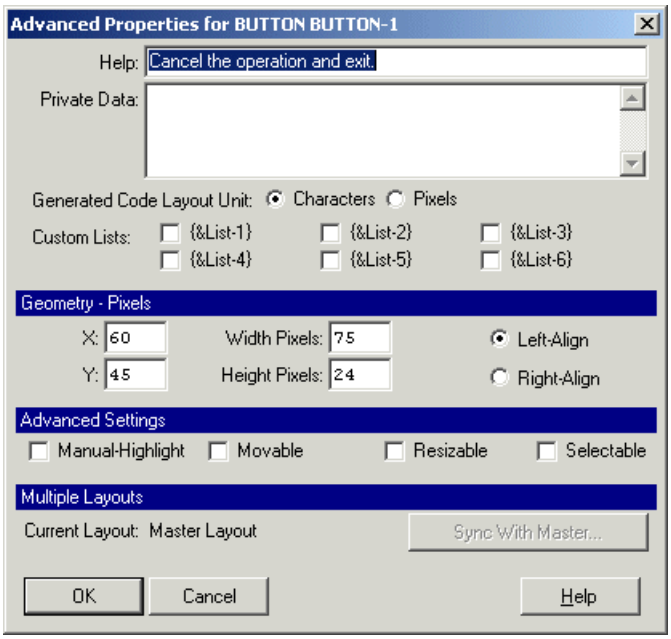
Within the AppBuilder, you can specify a help string for a field-level object in its **Advanced Properties** dialog box.

To define help text for a field-level object in the AppBuilder:

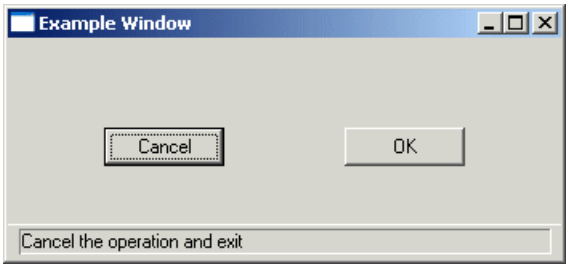
1. Open your window (.w) file in the AppBuilder.

   Note: Make sure that the status area of your window is enabled. To check this option, select the design window's title bar then click **Object Properties** from the AppBuilder toolbar. The property sheet for the window appears. Verify the **Status-Area** option is checked.

2. Double-click on a field-level object, such as a button. The property sheet for the object appears.
3. Click **Advanced** in the property sheet. The **Advanced Properties** dialog box appears.
4. Type a help message in the **Help** field. In this example, the message "Cancel the operation and exit." was added for a **Cancel** button:



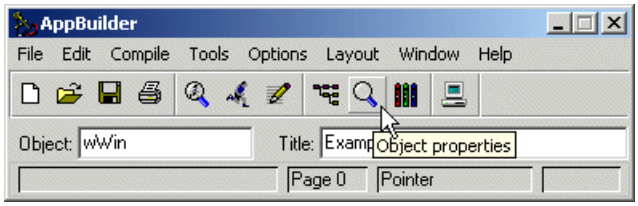
5. Click **OK** to close the **Advanced Properties** dialog box.
6. Click **OK** again to close the property sheet.
7. Save the window. The help string that you typed in the **Advanced Properties** dialog box becomes part of a `HELP` option of a `FORMAT` phrase in the `DEFINE FRAME` statement generated by the AppBuilder.
8. Run the window and select the object (in this example, the **Cancel** button). The help text appears in the status area of the window:



ToolTips

A ToolTip is a short string that appears in an enclosing rectangle when the user pauses the mouse pointer over a field-level widget. ToolTips are widely used in Windows applications. For example, ToolTips are often used to provide labels for toolbar buttons, as shown in the following figure.

Figure 42: ToolTip example



You can assign a ToolTip to any field-level object such as a button, combo box, editor, fill-in field, image, radio set, selection list, slider, text, and toggle box.

Implementing ToolTips with ABL

To implement ToolTips, you specify the ToolTip text in the TOOLTIP attribute of the associated object and set the TOOLTIPS attribute of the SESSION system handle to TRUE.

TOOLTIP attribute

Each object for which the TOOLTIP option is implemented has a run-time attribute, called TOOLTIP, established with read/write capabilities. If the TOOLTIP attribute is set to "" or the Unknown value (?), then no ToolTip is displayed for that object. The default is to not have a ToolTip. You can add a ToolTip to an object at any time.

Here is the syntax for the TOOLTIP option:

Syntax

```
[ TOOLTIP tooltip]
```

tooltip

A quoted string containing the text that displays when the user pauses the mouse pointer over the object.

The following code example shows how to specify the TOOLTIP attribute at run time:

```
Btn_OK:TOOLTIP = "Select this button to accept the information."
```

You can use the TOOLTIP option with the following ABL elements:

- DEFINE BROWSE statement
- DEFINE BUTTON statement
- DEFINE IMAGE statement
- DEFINE RECTANGLE statement
- VIEW-AS phrase for combo box, editor, fill-in, radio set, selection list, slider, text, and toggle box objects.

TOOLTIPS attribute

There is a session attribute called TOOLTIPS. The session default setting for TOOLTIPS is on (TRUE). To turn TOOLTIPS off for the session, set the TOOLTIPS session attribute to FALSE. For example:

```
SESSION:TOOLTIPS = FALSE
```

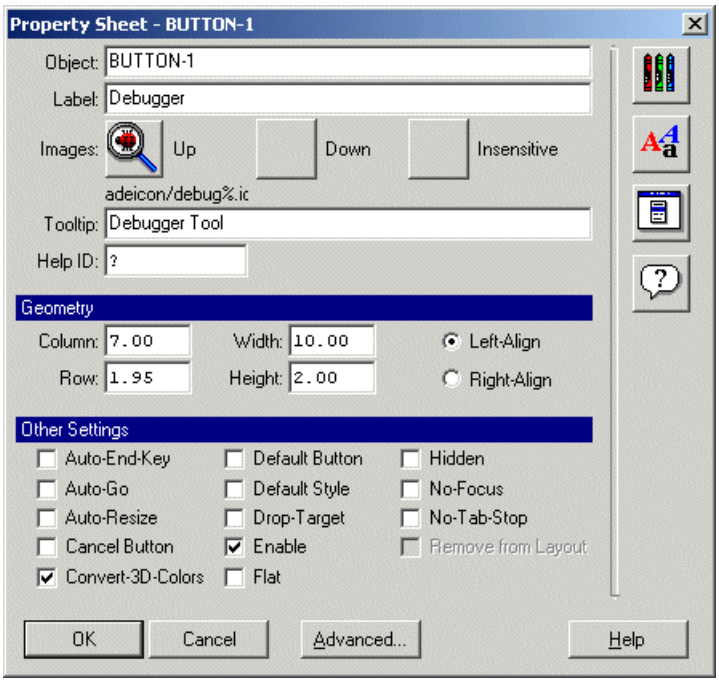
For more information on session attributes, see *ABL Reference*.

Implementing TOOLTIPS with the AppBuilder

The OpenEdge AppBuilder tool allows you to define `TOOLTIPS` by entering ToolTip text in an object's property sheet.

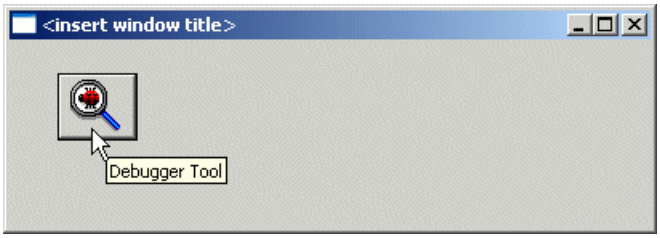
To define ToolTips using the AppBuilder tool:

1. Open a design window in the AppBuilder.
2. Select a field-level object (such as a button) in the design window, then click **Object Properties** to open its property sheet.
3. Type some help text in the **Tooltip** fill-in field:



Note: The **Help ID** field is not related to ToolTips. It is for specifying an identifier for the help topic (in a help file) associated with this object.

4. Click **OK**, then save the window.
5. Run the window and pass the mouse cursor over the object. The ToolTip appears in a rectangular box:



Online help systems

While field-level help in the form of status messages and ToolTips offers a first level of information for end users, the information it provides is limited. A help system can provide much larger amounts of information to the end user with a much more sophisticated delivery mechanism.

From the end user's perspective, there are two general ways to access information in a help system:

- **Context sensitivity** — Context-sensitive help is information that is primarily accessed "on demand." Typically, the end user clicks a help button in a dialog box, presses a help key (usually **F1**), or clicks the question mark icon to access a help topic that explains the current status or user interface of an application.
- **Navigation** — A help file can be opened in such a way that the end user is not immediately presented with a specific help topic. For example, the **Help Topics** item on the **Help** menu of an application window opens a help viewer that allows the end user to access help topics in several different ways.

  Help topics containing conceptual information or reference information are typically accessible only through navigation.

  Individual help topics can also contain links to other topics that provide related information in the form of definitions and other help topics. Also, help viewer windows usually implement features that allow end users to navigate among help topics in various ways.

In a help system, help information is stored in a help file that is external to the application's source code. A help author creates the help file, which is divided into chunks of information called help topics. An application programmer uses the `SYSTEM-HELP` statement to direct Windows to either display a specific help topic in a help viewer window, or allow the end user to navigate through the help topics via the help file's table of contents or search program.

While a help system should appear to end users as a seamless part of an OpenEdge application, the ABL `SYSTEM-HELP` statement actually calls a separate Windows application. For Microsoft HTML help (`.chm`) files it calls `hh.exe`, which launches the Microsoft HTML Help Viewer.

For information about what you need to create and run Microsoft HTML help applications, go to the Microsoft Developer's Network Web site (`msdn.microsoft.com`) and search for `HTML help`.

Because help information is stored in an external help file or files, a help system requires a coordinated effort of information design and application programming. The help author codes, compiles, and tests the help files, and the application programmer adds the appropriate ABL code to the application to invoke the help viewer and display the appropriate help topic when the end user requests help.

A help system consists of three elements:

- **Help information** — Text that assists users so that they can understand and use an application. The help information for an application resides in one or more help files and is displayed in chunks or units of information called help topics. Each help topic contains information about a single subject. For example, a help topic might describe a dialog box in your application, define a term, give instructions on how to perform a task, or describe an ABL language element. Each help topic is associated with a unique identifier, called a topic ID.

  Both types of help files are generated by compiling a set of help source files. The HTML Help (`.chm`) files are compiled HTML files. The Windows Help files (`.hlp`) are compiled binary files.

  For more information on how to organize and write help information, go to the Microsoft Developer's Network Web site (`msdn.microsoft.com`).

- **Help engine** — The executable file (`hh.exe` for HTML help or `winhelp32.exe` in Windows Help) displays help topics in a help viewer that allows the user to navigate among help topics in a help file.

  For more information about the help engines, go to the Microsoft Developer's Network Web site (`msdn.microsoft.com`).

- **Help calling interface** — The code in an application that allows users to access help information. It calls the help viewer and determines which help topic to display. Users can request and receive help information using a help keystroke, a help button, or a help menu.

The following sections describe in detail how to create a help calling interface for an OpenEdge application.

The SYSTEM-HELP statement

An OpenEdge application executes the Windows help engine (hh.exe or Winhlp32.exe) using the SYSTEM-HELP statement. Here is the syntax for the SYSTEM-HELP statement:

Syntax

```
SYSTEM-HELP file-string
   [ WINDOW-NAME window-name ]
   {   CONTENTS
     | CONTEXT int-expr
     | HELP-TOPIC string
     | KEY string
     | ALTERNATE-KEY string
     | POSITION X x Y y WIDTH dx HEIGHT dy
     | POSITION MAXIMIZE
     | QUIT
     | SET-CONTENTS int-expr
     | CONTEXT-POPUP int-expr
     | PARTIAL-KEY string
     | MULTIPLE-KEY char TEXT string
     | COMMAND string
     | FINDER
     | FORCE-FILE
     | HELP
   }
```

`file-string`

The file-string parameter is a character expression that specifies the pathname of a help file. If the file has a .chm extension (the extension for compiled Microsoft HTML Help files), the Microsoft HTML Help viewer is launched. If the file has a .hlp file extension, the Microsoft Windows Help viewer is launched.

`WINDOW-NAME` *`window-name`*

The *window-name* parameter is a character expression that evaluates to the primary or secondary window name as defined in the [WINDOWS] section of the help project file. If the window name is omitted, or if "main" is specified, the primary help window is used.

Note: This option is supported in Windows Help (.hlp files) only.

CONTENTS

For HTML Help, this option displays the Microsoft HTML Help viewer with the default topic in the content pane. Use the HELP-TOPIC option to specify the topic to display.

In Windows Help, this option displays the help topic defined as the contents in the [OPTIONS] section of the help project file.

Note: This option is supported for backward compatibility only.

CONTEXT *int-expr*

Displays the help topic that the context number identifies. You define context numbers in the [MAP] section of the help project file.

The *int-expr* parameter is the context number for the help topic.

HELP-TOPIC *string*

Displays a help topic in the content pane of the Microsoft HTML Help viewer.

The *string* parameter is a character expression that indicates the topic (.htm/.html file) within the compiled Microsoft HTML Help (.chm) file to display.

Note: This option is supported for HTML Help (.chm files) only.

KEY *string*

For HTML Help, this option displays the topic matching the string found in the keyword index. Use semicolons in the *string* parameter to delimit multiple keywords. If no match is found, Microsoft HTML Help displays the help viewer with the **Index** tab on top.

In Windows Help, this option displays the help topic matching the string found in the index keyword list. If there is more than one match, it displays the first topic containing the keyword. If there is no match or the string is omitted, a message is displayed indicating that the keyword is invalid. The *string* parameter is a character expression that evaluates to a keyword for the desired help topic.

ALTERNATE-KEY *string*

Displays a help topic matching the string found in the alternate keyword (Alink) index. The *string* parameter is a character expression that evaluates to a keyword in the alternate keyword index.

Note: This option is supported for HTML Help (.chm files) only. In Windows Help (.hlp files), see the MULTIPLE–KEY option.

POSITION X *x* Y *y* WIDTH *dx* HEIGHT *dy*

Positions an existing (already opened) help window as specified.

The *x* parameter is an integer expression that specifies the x coordinate for the help window.

The *y* parameter is an integer expression that specifies the y coordinate for the help window.

The *dx* parameter is an integer expression that specifies the width of the help window.

The *dy* parameter is an integer expression that specifies the height of the help window.

`POSITION MAXIMIZE`

Maximizes an existing (already opened) help window.

`QUIT`

Informs the help application that help is no longer required. If no other applications are using help, the operating system closes the help application.

`SET-CONTENTS` *`int-expr`*

Dynamically remaps the contents help topic from what is defined in the [OPTIONS] section of the help project file. When a `CONTENTS` call is made, the new contents help topic is displayed.

The *`int-expr`* parameter is the context number for the new contents help topic.

Note: This option is supported in Windows Help (`.hlp` files) only. This option is supported for backward compatibility only.

`CONTEXT-POPUP` *`int-expr`*

Displays the help topic in a pop-up window that the context number identifies. You define context numbers in the [MAP] section of the help project file. If a nonscrolling region exists in a help topic, only that region displays when you use the `CONTEXT-POPUP` option to display the topic.

The *`int-expr`* parameter is the context number for the help topic.

Note: This option is supported in Windows Help (`.hlp` files) only.

`PARTIAL-KEY` *`string`*

Displays the help topic matching the string found in the keyword list. In Windows, if there is more than one match, no match, or if the string is omitted, it displays the **Help Topics: Window Help Topics** dialog box with the **Index** tab on top.

The *`string`* parameter is a character expression that evaluates to a partial key for the desired help topic.

Note: This option is supported in Windows Help (`.hlp` files) only.

`MULTIPLE-KEY` *`char`* `TEXT` *`string`*

Displays the help topic matching a keyword from an alternate keyword table.

The *`char`* parameter is a character expression that evaluates to the single character keyword table identifier for the required table.

The *`string`* parameter is a character expression that evaluates to the keyword that is located in the keyword table.

Note: This option is supported in Windows Help (`.hlp` files) only. For HTML Help, see the `ALTERNATE-KEY` option.

`COMMAND` *`string`*

Executes a help macro.

The *string* parameter is a character expression that evaluates to the help macro to execute.

Note: This option is supported in Windows Help (.hlp files) only.

FINDER

Displays the **Help Topics: Windows Help Topics** dialog box, which contains an **Index** tab, a **Find** tab, and optionally a **Contents** tab, with the most recently used tab displayed on top.

If a **Contents** tab file (.cnt file) is present when you initially call the **Help Topics: Windows Help** dialog box, then the **Contents** tab displays on top. However, if a .cnt file is not present, then the dialog box displays with the **Index** tab on top; the **Contents** tab is not available.

Note: This option is supported in Windows Help (.hlp files) only.

FORCE-FILE

Ensures that the correct help file is open and displayed.

Note: This option is supported in Windows Help (.hlp files) only.

HELP

Displays the contents of the OpenEdge Help-on-Help file. In Windows, HELP displays the **Help Topics: Windows Help Topics** dialog box.

Note: This option is supported in Windows Help (.hlp files) only.

Accessing online help from the menu bar

A common method of requesting help from an application is by selecting help items from the application menu bar. The menu bar usually includes a help pull-down menu and contains menu items, such as:

- **Help Topics** — Instruct the help engine to display the HTML Help Viewer for the specified help file
- **About or Version** — Provide application copyright and version number information

The CHOOSE event occurs when a user chooses an option from a menu. In ABL, you use a CHOOSE event to code a trigger for menu items.

The following code example demonstrates a help trigger that executes when the user selects a menu item from the application's menu bar to access the help engine:

```
ON CHOOSE OF MENU-ITEM menu_help_contents DO:
  SYSTEM-HELP myhelp.chm CONTENTS.
END.
```

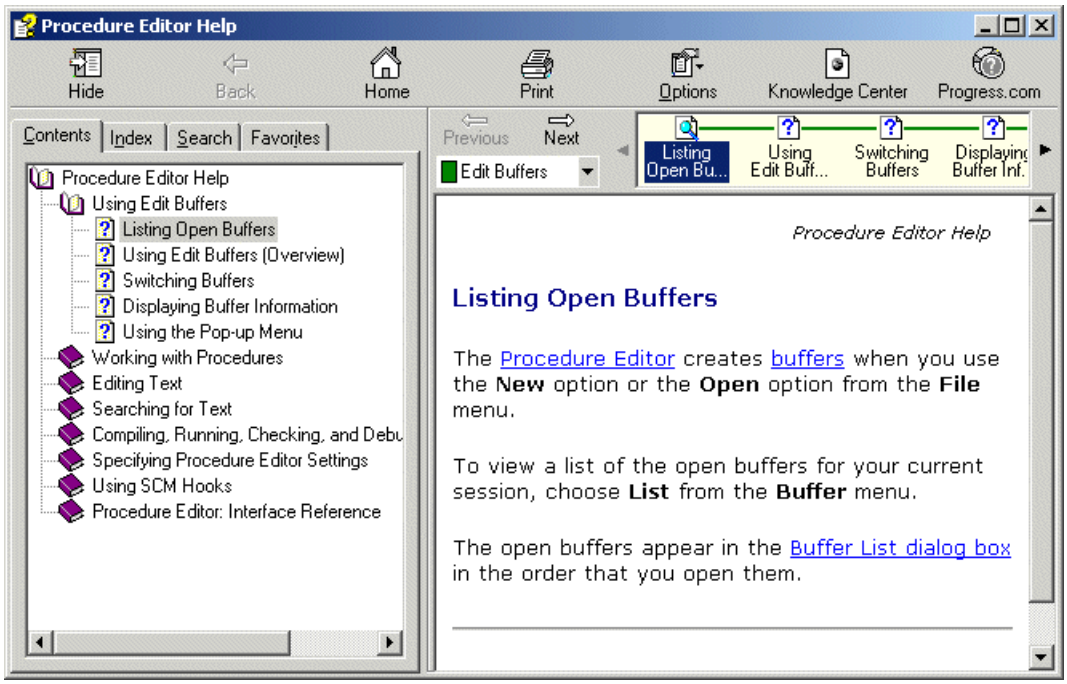
Calling the Help Viewer

Using the CONTENTS option of the SYSTEM-HELP statement displays the HTML Help Viewer for the specified help file. You frequently use this for a Help Topics call from the help menu. The following code example demonstrates a **Help Topics** call:

```
ON CHOOSE OF MENU-ITEM Help_Topics DO:
  SYSTEM-HELP myhelp.chm CONTENTS.
END.
```

For example, when the user chooses the **Help Topics** from the **Help** menu in the Procedure Editor's main window, the Help Viewer appears, as shown in the following figure.

Figure 43: HTML Help Viewer showing Procedure Editor Help



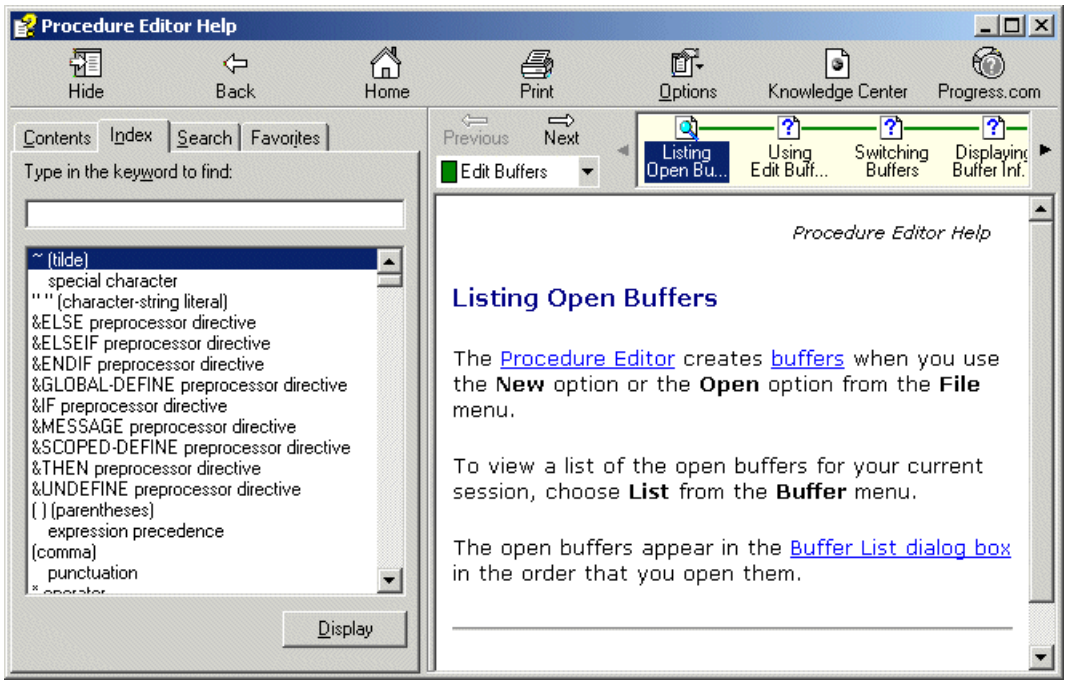
Calling the Index function of the Help Viewer

Like the **Help Topics** call, it is common to call the **Index** function of the Help Viewer from the **Help** menu. The following pseudocode calls the Help Viewer with the **Index** tab on top:

```
ON CHOOSE OF Button_Search IN FRAME frame_1 DO:
   SYSTEM-HELP myhelp.chm KEY "".
END.
```

The following figure shows the Help Viewer for the Procedure Editor help file with the **Index** tab on top.

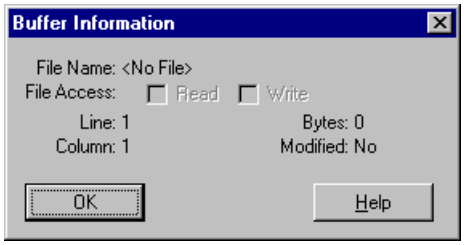
Figure 44: Help Viewer with Index tab on top



Calling online help with a Help button

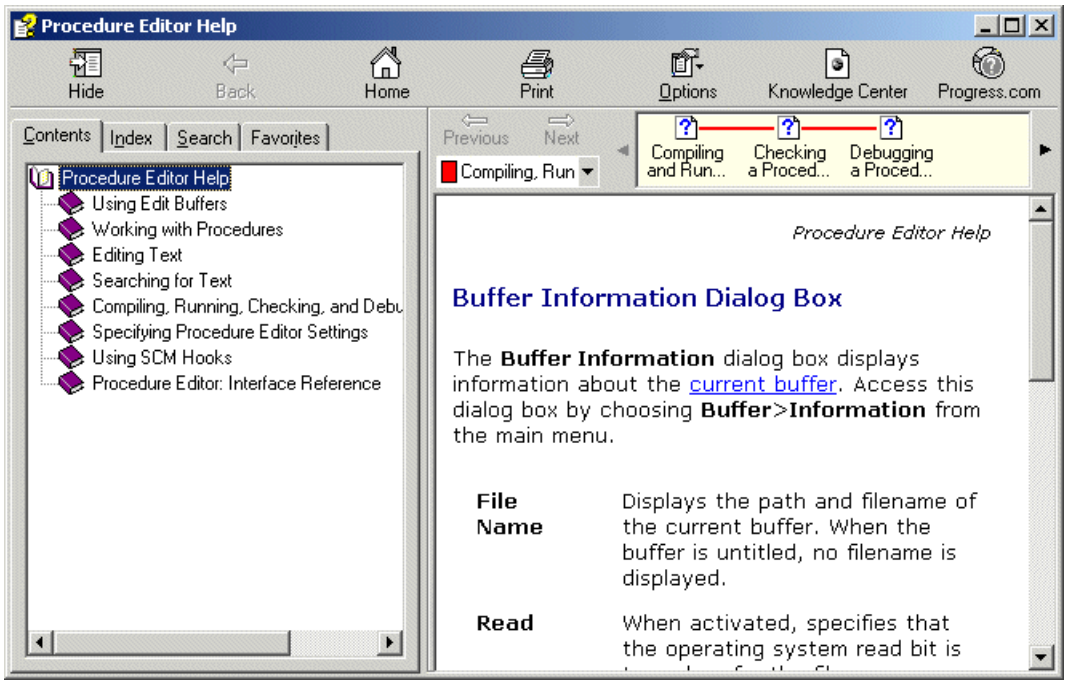
Your application should provide context-sensitive help when a user wants to learn about the purpose of a particular window, dialog box, or widget. You can code the OpenEdge application to display a **Help** button, as shown in the following figure. When the user clicks the button, the Help Viewer displays the help topic for the current window or dialog box.

Figure 45: Dialog box with a Help button



For example, when the user clicks the **Help** button in the Procedure Editor **Buffer Information** dialog box, the user sees context-specific help for that dialog box, as shown in the following figure.

Figure 46: Context-sensitive help for the Buffer Information dialog box



As with menu items, you attach triggers to buttons with the CHOOSE event. In this example, a help trigger is attached to the **Help** button. In the following code example, a help trigger executes when the user clicks a **Help** button to access help information from an application window or dialog box:

```
ON CHOOSE OF b_help IN FRAME x DO:
  SYSTEM-HELP myhelp.hlp CONTEXT 49154.
END.
```

You can write a help trigger for a field-level widget, a frame, a dialog box, a window, or an application. The following code example demonstrates a help trigger that executes when the current application window has input focus and the user presses **HELP**:

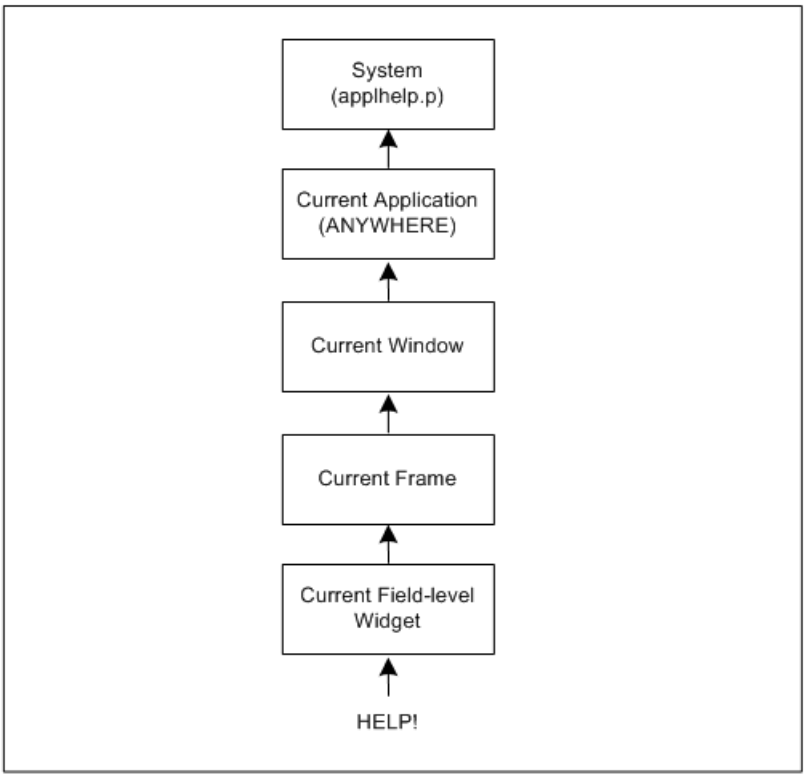
```
ON HELP OF WINDOW-1 DO:
  SYSTEM-HELP myhelp.hlp CONTEXT 49154.
END.
```

Accessing online help with the help key

When a user presses the designated help key (usually **F1** in Windows applications), a HELP event goes to the field-level widget with input focus in the current frame. If there is no trigger on the field-level widget, the HELP event goes to the current frame. The HELP event continues to move to the next level in the widget hierarchy, as shown in the following figure. If the help event does not find an associated widget, it runs applhelp.p, the OpenEdge help calling interface. This procedure, applhelp.p, in *<install-dir>*\src\, displays the message, "No application help is available."

The following figure depicts the behavior of the ABL HELP event.

Figure 47: HELP Event Hierarchy



Quitting help when exiting the application

You can terminate the help viewer with the QUIT parameter. As a matter of sound programming practice, your OpenEdge application should execute the following line of code when it terminates:

```
SYSTEM-HELP myhelp.hlp QUIT.
```

If the help engine is running when the user terminates the OpenEdge application, then the Help Viewer also terminates.

13

Named Pipes

In the UNIX and Windows environments, you can establish interprocess communications (IPC) between a non-OpenEdge application (such as a C program or commercial software package) and a OpenEdge session using named pipes. This facility provides a capability similar to Dynamic Data Exchange (DDE) in Windows, though it works very differently. Named pipes provide a general exchange mechanism for text data. Any data you can access as a character string within OpenEdge, you can read or write to a named pipe.

OpenEdge accesses named pipes already created on your UNIX or Windows systems. OpenEdge does not create a named pipe itself. Named pipes are only valid on UNIX or Windows systems. This can pose a portability issue.

From ABL (Advanced Business Language), named pipes look and act like operating system files. To exchange data, the OpenEdge application reads or writes to a named pipe, just as it does to a file. However, instead of a file at the end of the pipe, the non-OpenEdge application reads or writes data to the OpenEdge application.

For details, see the following topics:

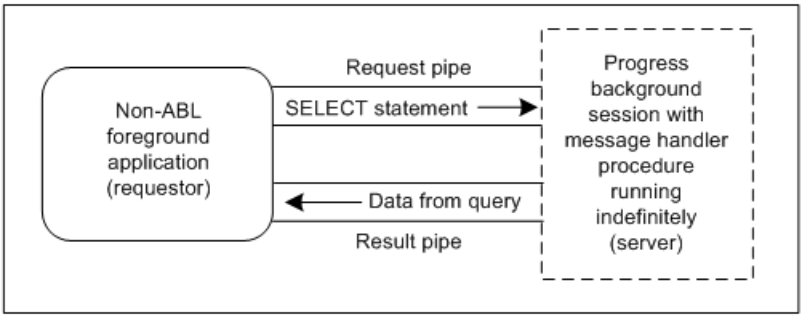
- Overview of named pipes with ABL
- UNIX named pipes
- Windows named pipes

Overview of named pipes with ABL

Named pipes provide a general exchange mechanism for text data, and there is no practical limit to the types of data you can exchange using them. Any data you can access as a character string within ABL, you can read or write to a named pipe. For example, you can use named pipes to issue SQL requests to an OpenEdge session from within a spreadsheet program and receive the resulting data in the spreadsheet. As such, you can use named pipes to implement some of the capabilities provided by ABL Host Language Call Interface (HLC).

The following figure shows a typical named pipe scenario.

Figure 48: Typical named pipe scenario



The *message handler procedure* acts as a server for a non-OpenEdge application acting as requestor. The server reads each incoming request, processes it, and returns the results through a second named pipe. The messages can contain SQL statements, ABL statements, procedure names, or anything that your message handler procedure can manage.

Access from ABL

ABL accesses named pipes already created on your system; ABL does not create named pipes itself. ABL treats a named pipe the same way as it treats a text file. ABL statements `INPUT FROM`, `OUTPUT TO`, `DISPLAY`, `SET`, `EXPORT`, and `IMPORT` all access named pipes and files identically.

Named pipes and files

Named pipes combine the features of files and unnamed pipes. Like a file, a named pipe has a name and any process with appropriate permissions can open it to read or write. Thus, unrelated processes can communicate over a named pipe. Like an unnamed pipe, a named pipe behaves like a first in/first out (FIFO) queue. A reading process reads and removes from the pipe the first unit of data written to the pipe that has not been read.

Uses for named pipes

The scenario in Figure 48: Typical named pipe scenario on page 466 illustrates important core concepts, but it is a relatively simple example of what you can do with named pipes. For example, you can design your message handler procedure to handle requests from more than one non-OpenEdge application user at a time. Another idea is to design a message handler that manages multi-line requests as well as single-line requests. This makes it possible for the requests to include, for example, ABL `FOR EACH` blocks.

Operational characteristics of named pipes

Once opened, named pipes act more like unnamed pipes than files. Data written to the named pipe is read in FIFO order. Once data written to a named pipe is read, it is removed from the named pipe. Also, the operating system regards individual reads and writes as unbreakable (atomic) units and issues them one at a time, unless the amount read or written exceeds the capacity of the named pipe. The capacity of a named pipe is the same as the capacity of an unnamed pipe. (The capacity of an unnamed pipe depends on the implementation; on UNIX environments, however, the amount is always 4,096 bytes or greater).

I/O synchronization

ABL accesses named pipes using unbuffered I/O. This means that processes that read from and write to the same named pipe synchronize their reads and writes with each other. In ABL, this is true for opening as well as reading and writing named pipes. When an ABL process opens a named pipe for input, it blocks (waits) until another process opens the same named pipe for output. The reverse is also true—when an ABL process opens a named pipe for output, it blocks until another process opens the same named pipe for input.

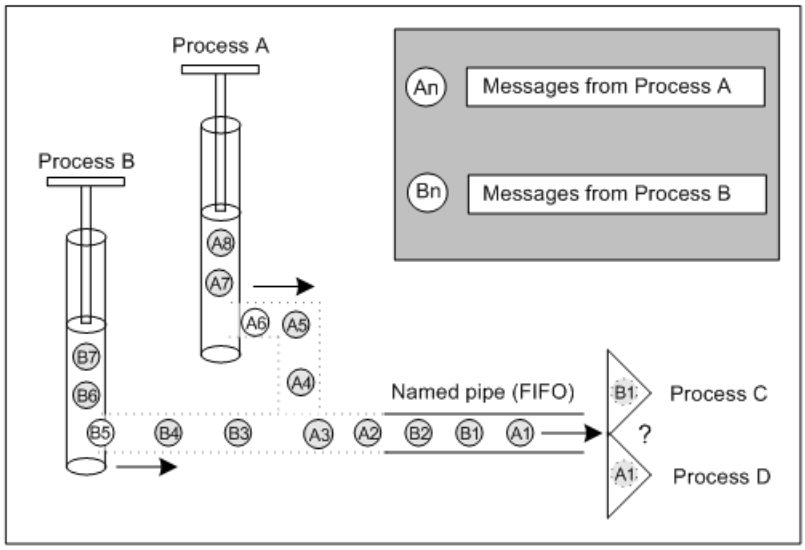
When a process writes to a named pipe, the process blocks until another process reads from the named pipe. Similarly, when a process attempts to read from a named pipe, but there is nothing to read, the process blocks until something is written to the named pipe.

Message interleaving

If multiple processes write messages to the same named pipe, the messages might be interleaved (mixed). However, as stated earlier, individual message reads and writes are atomic.

For example, suppose there are two processes, Process A and Process B. Each process writes several messages to the same named pipe. As they are written, some of the messages from Process A might become mixed with messages from Process B. However, an individual message cannot be interrupted by another message, since the messages are atomic. The following figure illustrates this example.

Figure 49: Writing messages to a named pipe



Also, note that if two processes (Process C and Process D) simultaneously read from the same named pipe, they receive messages from both Process A and Process B in order of transmission, but whether Process C or D receives a particular message might be uncertain. In the above figure, either Process C or Process D can receive message A1 or B1, if the processes read the pipe at the same time. The actual messages received by which process depend on the state of the system at the time of input. In other words, all applications that use a named pipe must establish a mutual protocol for effective cooperation.

Advantages and disadvantages of named pipes

A major advantage of using named pipes is that they provide a useful way to send one-line requests to an OpenEdge background session running a message handler procedure. Multiple users can send requests through the same named pipe and each request is removed from the pipe as it is received. In addition, the message handler procedure can loop indefinitely looking for input because it blocks (waits) until there is something to read. Finally, output through named pipes is more efficient than writing a complete response to an ordinary file, closing the file, and then informing the recipient that the results are available. The receiving process can read the result through a named pipe as soon as it is written.

A disadvantage of named pipes is that multiple processes cannot use a single named pipe to send or receive multi-line messages, unless you define a more complex protocol to control message interleaving. Also, although synchronizing named pipe input and output is helpful in some situations, it is a problem in others. For example, if the message handler procedure running in the background OpenEdge session starts returning results to an output named pipe, and for some reason the requestor is not ready to read the results, the message handler cannot move on to read the next request.

UNIX named pipes

These topics provide information to get you started using named pipes with ABL on a UNIX system. You can find more information on UNIX named pipes in any of the books on advanced UNIX programming available in the public domain.

To use UNIX named pipes with ABL:

1. Create the named pipes using the UNIX `mknod` command in the command line or the `mknod( )` system call from within C.
2. Start an OpenEdge session in the background (batch mode) running a message handler procedure. The message handler procedure runs indefinitely, searching for input from one named pipe, running requests, and shipping output through a second named pipe. (You must supply the message handler procedure. For a sample message handler procedure, see UNIX named pipe examples on page 470.)
3. Run your non-OpenEdge application. From within the application, issue messages through the first named pipe in Step 1 to the background OpenEdge session, and receive replies through the second named pipe.

Once you create a named pipe, you can access it as if it were a text file. Thus, the only requirement for a non-OpenEdge application to communicate with ABL via named pipes is that the application be able to write to and read from text files. Also, it is helpful for the application to have facilities for processing returned results (for example, string handling functions, buffers, etc.).

The following sections describe how to:

- Create a UNIX named pipe
- Delete a UNIX named pipe
- Use a UNIX named pipe between OpenEdge and non-OpenEdge applications

Note: Named pipes might be implemented on your system differently than described in the next section.

Creating a named pipe

To create a UNIX named pipe, use the `mknod` command on the command line or the `mknod( )` system call from a C program. The two techniques produce the same results. The examples in these topics use the command-line technique.

Once you create a named pipe, its characteristics are similar to an ordinary file. For example, it is located in a directory, has a pathname, and exists until you delete it.

The `mknod` command has more than one form. This is the syntax for the form that creates a named pipe:

Syntax

```
mknod named-pipe-identifier p
```

The *`named-pipe-identifier`* is the pathname of the named pipe you want to create.

For example, to create a named pipe called `mypipe` in the current directory, type the following command:

```
mknod mypipe p
```

The following C function shows how to use the `mknod()` system call to create a named pipe:

```
int mkfifo(path) /* make FIFO */
char *path;
{
  return(mknod(path, S_IFIFO | 0666, 0));
}
```

For more information on `mknod` or `mknod()`, see your UNIX system documentation.

Deleting a named pipe

To delete a named pipe on UNIX, use the `rm` command.

For example, to delete the named pipe `mypipe` in the working directory, type the following command:

```
rm mypipe
```

From within a C program, use the `unlink()` system call. For more information on `unlink()`, see your system documentation.

Accessing a named pipe within ABL

To access a named pipe from ABL, open it for input or output using the `INPUT FROM` and `OUTPUT TO` statements. For example, the following line of ABL code opens the previously created named pipe `inpipe` for input:

```
INPUT FROM inpipe NO-ECHO.
```

After invoking this statement, all input statements that use the unnamed stream, such as `SET` or `IMPORT`, take their input from `inpipe`.

UNIX named pipe examples

The following examples show different uses of named pipes. To provide a simple example of how named pipes operate, the first example shows how to use the shell to create a named pipe, send a message to it, and read the message back. The second example shows how to use named pipes with ABL.

Example 1: creating and using a named pipe from the shell

In the `i-pipex1` example, the `cat` command sets up a message handler routine and the `echo` command acts as the requestor.

i-pipex1

```
# Named Pipe Example 1.
#
# Create named pipe...
mknod trypipe p
# Open named pipe and read message...
cat trypipe &
# Write message...
echo "This is a message!" > trypipe
# Delete pipe...
rm trypipe
```

To try this example, run the shell script, `i-pipex1`. This script performs the following actions:

1. The `mknod` command creates a named pipe called `trypipe`.
2. The `cat` command opens `trypipe` for reading. It blocks because `trypipe` has not yet been opened for writing. Notice that an ampersand (`&`) is present at the end of the `cat` command; this runs the `cat` command as a background process.
3. The `echo` command opens `trypipe` for writing and writes a message. The `cat` command, blocked until now, resumes execution, and the message appears on the display.
4. The `rm` command deletes `trypipe`.

Example 2: using a named pipe with ABL

Before working with the following procedures, create a copy of the `demo` database with the PRODB utility:

```
prodb demo demo
```

This example shows a simple user program that sends one line requests to an ABL message handler routine running in the background, and displays the results. The example consists of four files:

1. A script called `i-pipex2` that runs the example:

I-pipex2

```
# Named Pipe Example 2.
#
# Create named pipes...
mknod inpipe p
mknod outpipe p
# Start OpenEdge background session with i-pipex2.p running...
bpro demo -1 -p i-pipex2.p
# Run executable i-asksql...
i-asksql
# Terminate OpenEdge background session...
echo "outpipe \"quit\"" > inpipe
cat outpipe
# Delete named pipes...
rm inpipe
rm outpipe
```

2. The message handler procedure, `i-pipex2.p`:

i-pipex2.p

```
/* Target variable for the request:*/
DEFINE VARIABLE sql-stmt AS CHARACTER NO-UNDO FORMAT "x(220)".
/* Holds the output file or FIFO: */
DEFINE VARIABLE out-pipe AS CHARACTER NO-UNDO FORMAT "x(32)".

/* Do forever: */
REPEAT:
  /* Set up to read from in-FIFO named "inpipe". */
  INPUT FROM inpipe NO-ECHO.
  /* For each request received: */
  REPEAT:
    /* Get the output name and the request. */
    IMPORT out-pipe sql-stmt.
    /* Set up to write results. */
    OUTPUT TO VALUE(out-pipe) APPEND.
    /* Pass SQL request to sub-proc. */
    RUN i-do-sql.p sql-stmt.
    OUTPUT CLOSE.
    /* This loop ends when the in-FIFO is empty. Just reopen it and wait
       for the next request. */
  END.
END.
```

3. A subprocedure, `i-do-sql.p`:

i-do-sql.p

```
/* This program consists of a single line of code. */
{1}
```

4. A C source file called `i-asksql.c` that implements the requestor:

i-asksql.c

```
#include <stdio.h>
#include <fcntl.h>

main()

{
#define  LEN     250
        char   result[LEN];
        int     fdi, fdo, nread;
        char   request[LEN+8]; /* 8 for "outpipe " + punctuation */
        char   *ptr;
        int     validq, i;
        fdi = open("inpipe", O_WRONLY);
        if (fdi < 0)
        { printf("Error on inpipe open\n"); exit(1);}

        strcpy(request, "outpipe \""); /* request starts with 'outpipe "' */

        while (1)
        {
          printf("\n\nEnter your request (type [RETURN] to exit):\n");
          ptr = request+9;
          nread = read(0, ptr, LEN);
          if (nread < 2)
              exit(0);
          else
          {
            validq = 1;             /* valid query? */
            for (i = 9; i<nread+9; i++)
                if (request[i] == '\"')
                { printf("Use only single quotes in queries.\n");
                  validq = 0;
                  break;
                }

 if (! validq) continue;
            ptr += nread-1;
            *ptr++ = '\"';
            *ptr++ = '\n';
            *ptr++ = '\0';
            write(fdi, request, strlen(request));
            sleep(1);
            fdo = open("outpipe", O_RDONLY);
            if (fdo < 0)
            { printf("Error on outpipe open\n"); exit(1);}

            while ((nread = read(fdo, result, LEN)) != 0)
            {
              result[nread] = '\0';
              printf("%s", result);
            }
            close(fdo);
          }
        }
}
```

To prepare and run the example:

1. Use `cc` to compile and link the requestor source `i-asksql.c` to produce the executable `i-asksql`, as shown:

```
cc i-asksql.c -o i-asksql
```

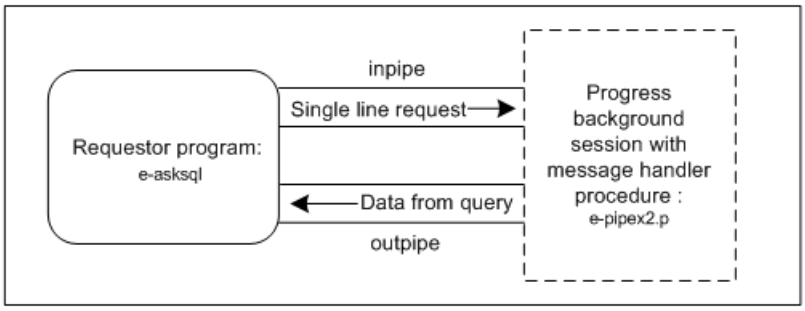
2. Execute the `i-pipex2` script to run the example:

```
i-pipex2
```

The `i-pipex2` script performs the following actions:

1. It uses `mknod` to create two named pipes: `inpipe` and `outpipe`. Named pipe `inpipe` carries requests from the requestor to the message handler routine. Named pipe `outpipe` carries results back in the opposite direction, from the message handler to the requestor. The following figure illustrates this process.

   Figure 50: Named pipes and ABL



2. It starts a background OpenEdge session that runs the message handler procedure, `i-pipex2.p`, with the demo database. The following line in `i-pipex2.p` opens the named pipe `inpipe` for input:

```
INPUT FROM inpipe NO-ECHO.
```

   Notice that ABL accesses named pipes in exactly the same way as UNIX text files. At this point, `i-pipex2.p` blocks until a requestor process opens `inpipe` for output.

3. After starting the ABL message handler, the script starts the requestor, `i-asksql`, which opens `inpipe` for output using the following statements:

```
int fdi, fdo, nread;
. . .
fdi = open("inpipe", O_WRONLY);
```

4. As `i-asksql` opens `inpipe`, `i-pipex2.p` unblocks and blocks again as it attempts to read a message from `inpipe`. The message handler procedure, `i-pipex2.p`, expects single-line requests from requestors, and can handle more than one requestor. This is the syntax for message handler requests:

Syntax

```
output-pipe-nameSQL-statement
```

Each request contains the name of the named (*output-named-pipe*) pipe from which the requestor expects to receive results and an SQL statement (*SQL-statement*) surrounded by double quotes (" "). The message handler procedure reads these messages with the following statement:

```
IMPORT out-pipe sql-stmt.
```

Each requestor must specify a unique value for *output-pipe-name*, or results might be intermixed. (Using the requestor's PID number as part of the name ensures uniqueness. However, for that to work, the requestor probably has to create its own named pipe using the `mknod( )` system call.) Note that for this example, the requestor, `i-asksql`, uses the existing named pipe `outpipe` created by the script.

5. As the message handler waits for input, the requestor displays the following prompt:

```
Enter your request (type [RETURN] to exit):
```

You can enter a one-line SQL query like the following `SELECT` from the demo database:

```
SELECT name FROM customer.
```

6. The requestor constructs a message from the name of the output pipe (`outpipe`, in the example) and the contents of your query, and writes the message to `inpipe`, as in the following statements from `i-asksql.c`:

```
char request[LEN+8]; /* 8 for "outpipe " + punctuation */
. . .
write(fdi, request, strlen(request));
```

7. As the message handler receives (and removes) the message from `inpipe`, it unblocks and opens the output pipe named in the message with the following statement:

```
OUTPUT TO VALUE(out-pipe).
```

8. The message handler blocks again, waiting for the requestor to open the same pipe for input (to receive the query result), as in the following statements from `i-asksql.c`:

```
int fdi, fdo, nread;
. . .
fdo = open("outpipe", O_RDONLY);
```

9. The ABL message handler then continues to compile and run the SQL query using the following statement:

```
RUN i-do-sql.p sql-stmt.
```

As the query generates output, ABL writes it one line at a time to the named pipe specified by `outpipe`. The requestor reads each line as it is written to the output pipe, as in the following statements from `i-asksql`. In the example, the requestor also writes each line to its standard output:

```
char result[LEN];
int fdi, fdo, nread;int fdi, fdo, nread;
. . .
while ((nread = read(fdo, result, LEN)) != 0)
{
    result[nread] = '\0';
    printf("%s", result);
}
```

Note: If there is no output, you might have entered your SQL statement incorrectly. This causes `i-pipex2.p` to terminate. To trap this type of error, write the SQL statement to a file instead of to a named pipe, then compile the file. If the compilation is successful, run it.

Note that although the query procedure, `i-do-sql.p`, contains only the single procedure parameter, `{1}`, you can extend it with formatting statements to avoid having to include these in each query, as in the following examples:

```
{1} WITH NO-LABELS.
```

```
{1} WITH EXPORT.
```

10. The example requestor, `i-asksql`, continues to prompt for queries, repeating Actions 5 through 9, until you press **RETURN** with no additional input.

11. After the requestor terminates, the `i-pipex2` script terminates the ABL background process with the following commands:

```
echo "outpipe \"quit\"" > inpipe
cat outpipe
```

The first command sends the ABL `QUIT` statement to the message handler (instead of an SQL statement). The second command takes the place of Action 8, originally handled by the requestor. The requestor does not send the `QUIT` to terminate the ABL background process so that multiple copies of the requestor—each with its own output pipe—can run without affecting the message handler. It is necessary because a process blocks until a named pipe it opens for writing is opened for reading (see Operational characteristics of named pipes on page 467). In this case, the message handler opens named pipe `outpipe` for writing, and cannot execute `QUIT` until the `cat` command opens `outpipe` for reading.

12. The `i-pipex2` script uses the `rm` command to remove the named pipes that it created.

Windows named pipes

OpenEdge supports named pipes in the Windows environment. In general, named pipes in the Windows environment behave similarly to UNIX named pipes. Some differences are:

- You cannot create Windows named pipes from the command line.
- Windows named pipes have different C language interfaces.

Accessing Windows named pipes

To access a Windows named pipe, you create it, connect it, read it, write it, and close it. The following table lists these tasks and their C and ABL equivalents.

Table 50: Using C and ABL to access Windows named pipes

Task	C	ABL
Create	`CreateNamedPipe()`	None
Connect	`ConnectNamedPipe()`	None
Read	`ReadFile()` `FlushFileBuffers()`	`INPUT FROM`
Write	`WriteFile()`	`OUTPUT TO`
Close	`CloseHandle()`	`INPUT CLOSE` `OUTPUT CLOSE`

As the above table shows, C lets you create, connect, read, write, and close Windows named pipes, while ABL lets you read, write, and close them.

Actually, ABL lets you perform all the tasks in the table if you use ABL's access to DLLs to call into `kernel32.dll`, which contains all the C functions in the table.

Linking OpenEdge and non-OpenEdge processes using Windows named pipes

You can link OpenEdge and non-OpenEdge processes using Windows named pipes, just as you can using UNIX named pipes. The resulting application consists of:

- A 32-bit OpenEdge Windows client running an ABL application in the background
- A non-OpenEdge program

The OpenEdge program reads, writes, and closes one or more Windows named pipes. The non-OpenEdge program creates, connects, reads, writes, and closes the named pipe or pipes.

Building and running the sample Windows named pipes application

This update describes a sample application consisting of an ABL program and a C program that communicate through a Windows named pipe. Both programs can read and write the named pipe. The ABL program accesses the `sports2000` database. When you run the application, you must tell one program to write the named pipe and the other program to read it.

If you tell the C program to write and the ABL program to read:

The C program...	While the ABL program...
• Solicits a customer number from the user • Writes the customer number to the named pipe	• Reads the named pipe, getting the customer number the user entered • Retrieves the row of the customer with the specified customer number • Displays the row of the customer table

If you tell the ABL program to write and the C program to read:

The ABL Program...	While the C program...
• Reads the entire customer table • Writes the name of each customer to the named pipe	• Reads the named pipe into a buffer • Displays the number of bytes read and the contents of the buffer

Coding the ABL program

The ABL program:

- Assumes that the C program creates and connects the named pipe
- Refers to the named pipe using the name the C program specifies
- Uses the `INPUT FROM` statement to read the named pipe
- Uses the `OUTPUT TO` statements to write the named pipe
- Uses the `INPUT CLOSE` and `OUTPUT CLOSE` statements to close the named pipe

The `i-ablpip.p` ABL program demonstrates reading and writing a Windows named pipe, `custpipe`.

i-pipex1

```
/* i-ablpip.p */
/* Reads and writes a Windows named pipe */

/* 1. Define buttons */
DEFINE BUTTON bWrite LABEL "Write to Pipe".
DEFINE BUTTON bRead  LABEL "Read from Pipe".
DEFINE BUTTON bQuit  LABEL "Quit".

/* 2. Define form */
FORM SKIP(5)
  SPACE(5) bWrite SPACE(5) bRead SPACE(5) SKIP(1)
  SPACE(18) bQuit SKIP(5)
  WITH FRAME f TITLE "Pipe Test".

/* 3. Define write trigger */
ON CHOOSE OF bWrite IN FRAME f DO:
  OUTPUT TO \\.\pipe\custpipe APPEND.
  FOR EACH Customer NO-LOCK:
    DISPLAY Customer.Name.
  END.
  OUTPUT CLOSE.
END.

/* 4. Define read trigger */
ON CHOOSE OF bRead IN FRAME f DO:
  DEFINE VARIABLE ix AS INTEGER NO-UNDO.

  INPUT FROM \\.\pipe\custpipe.
  SET ix.
  INPUT CLOSE.
  FIND Customer NO-LOCK WHERE Customer.CustNum = ix.
  DISPLAY Customer WITH 2 COLUMNS FRAME y OVERLAY TITLE "Customer Info".
END.

/* 5. Define quit trigger */
ON CHOOSE OF bQuit IN FRAME f DO:
  APPLY "window-close" TO CURRENT-WINDOW.
END.

/* 6. Enable all objects, then wait on a close event */
ENABLE ALL WITH FRAME f.
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

The `i-ablpip.p` program:

1. Defines three buttons, labeled "Write to Pipe," "Read from Pipe," and "Quit."
2. Defines a form to contain the buttons.
3. Defines a trigger for the **Write to Pipe** button. The trigger redirects output to named pipe `custpipe`, displays (to named pipe `custpipe`) the name of each `Customer` in the `sports2000` database, and closes the named pipe.

   In an `OUTPUT TO` statement, `\\.` means the current machine. To communicate with remote machine "pcdev68," for example, use `\\pcdev68`. This follows Uniform Naming Conventions (UNC).

   The `OUTPUT TO` statement uses the `APPEND` option, which causes ABL to open the named pipe without first creating it. This is necessary because ABL cannot create named pipes.
4. Defines a trigger for the **Read to Pipe** button. The trigger defines an integer data item, reads named pipe `custpipe`, assigns the value read (a customer number) to the integer data item, closes the named pipe, retrieves the row of the customer table with the specified customer number, and displays the columns of the row.

The `INPUT FROM` statement assumes that the pipe exists and that another process writes to it. The `INPUT FROM` statement blocks until the other process writes to the named pipe.

5. Defines a trigger for the **Quit** button.
6. Enables all objects in the frame and waits on a close event.

Coding the C program

The non-OpenEdge program creates, connects, reads, writes, and closes the Windows named pipe or pipes.

The `i-cpipe.c` C program demonstrates creating, connecting, reading, writing, and closing Windows named pipe `custpipe`.

i-cpipe.c

```
/* C program that reads and writes a Windows NT named pipe */
/* 1. Declare include files */
#include <windows.h>
#include <stdio.h>
#include <wincon.h>
#include <winerror.h>
#include <conio.h>
void main()
{
/* 2. Define automatic data items */
  HANDLE hPipe;
  char buffer[4096];
  DWORD dwBytesRead;
  BOOL bRet;
  int iPipeType;
  int iLen;
/* 3. Make window title meaningful */
  /* 4. Create the named pipe */
  printf("Creating Named Pipe custpipe\n");
  hPipe = CreateNamedPipe("\\\\.\\pipe\\custpipe",
    PIPE_ACCESS_DUPLEX,
    PIPE_TYPE_BYTE | PIPE_READMODE_BYTE | PIPE_WAIT,
    1,
    0,
    0,
    NMPWAIT_USE_DEFAULT_WAIT,
    NULL);
  if (hPipe == INVALID_HANDLE_VALUE)
  {
    printf("Error creating pipe, %ld\n", GetLastError());
    exit(GetLastError());
  }
  printf("custpipe created successfully\n");
/* 5. Solicit user input */
  do
  { printf("\nPress 1 for Read, 2 for Write: ");
    iPipeType = getch(); }
  while (iPipeType != '1' && iPipeType != '2');
/* 6. Connect the named pipe */
  printf("\nWaiting for connection...");
  if (ConnectNamedPipe(hPipe, NULL) == FALSE)
  {
    printf("Error connecting to named pipe, %ld", GetLastError());
    exit(GetLastError());
  }
  printf("and connection established\n");
```

```
/* 7. Read and write the pipe */
  switch(iPipeType)
  {
    case '1': /* read */
    while (TRUE)
    {
      bRet = ReadFile(hPipe, &buffer, 4096, &dwBytesRead, NULL);
      if (bRet == FALSE)
      {
        /* if the pipe went away, the ABL app did an OUTPUT CLOSE */
        if (GetLastError() == ERROR_BROKEN_PIPE)
        printf("Error reading from pipe, %ld\n", GetLastError());
        break;
      }
      printf("Data read = %ld %s\n", dwBytesRead, buffer);
    } /* end while */
    break;
  /*}  end switch */
case '2': /* write */
    while (TRUE)
      {
        printf(
           "Enter data to send, followed by ENTER: (Blank to exit) \n"
          );
        gets_s(buffer, sizeof(buffer));
        if (buffer[0] == 0)
          break;
        iLen = strlen(buffer);
        buffer[iLen++] = 0x0d; /* CR */
        buffer[iLen++] = 0x0a; /* LF */
        bRet = WriteFile(hPipe, &buffer, (DWORD) iLen, &dwBytesRead,
                    NULL);
if (bRet)
{          FlushFileBuffers(hPipe);
          printf("%ld bytes flushed to pipe\n", dwBytesRead);
        }
        else
          printf("Error writing to pipe: %d\n", GetLastError());
      } /* end while */
      break;
    } /* end switch */
/* 8. Close and exit */
  CloseHandle(hPipe);
  printf("custpipe closed successfully\nBye");
}
```

The `i-cpipe.c` program:

1. Declares include files.
2. Defines data items.
3. Calls an NT Console API function to give the window a more descriptive title.
4. Creates named pipe `custpipe`. The `PIPE_ACCESS_DUPLEX` flag makes the pipe read/write. The `PIPE_WAIT` flag makes the pipe synchronous. The program checks for errors and prints debugging messages here and throughout.
5. Solicits and accepts a value ("1" to read the pipe, "2" to write the pipe) from the user.
6. Connects the named pipe. This makes the pipe available to other applications, processes and threads.
7. Either reads the pipe and displays the number of bytes read along with the actual data, or else solicits a customer number, appends a carriage return and a line feed (CR/LF), writes the result to the named pipe, and flushes the buffers.

The program uses the same API calls for named pipes as for files. Named pipes are an integral part of the Windows file system, just as they are an integral part of UNIX file systems.

If the program did not append a CR/LF to the data it writes to the named pipe, the ABL program's SET input statement would wait for a line terminator or EOF marker, which does not ordinarily appear until the pipe is closed.

The two programs use a quick and dirty hack to signal "named pipe EOF." The ABL program closes the named pipe with the `OUTPUT CLOSE` statement, which causes the C program's ReadFile() call to raise a `BROKEN_PIPE` error, which the C program interprets as "named pipe EOF."

8. Closes the named pipe and exits.

Running the application

Before you run the application, you must compile and link the C program. This procedure assumes that the name of the C executable is `i-cpipe`.

To run the application:

1. In Windows, open a command window, which resembles an MS-DOS box.
2. Enter `prowin32 -p i-ablpip.p -1 sports2000` to start the OpenEdge process.
3. Enter `i-cpipe` to start the non-OpenEdge process.
4. In the C program, type `1` to select reading a named pipe.
5. In the ABL program, select **Write to Pipe**. The following occurs:
   - The ABL program writes the customer name in each row of the customer table to the named pipe.
   - The C program reads the named pipe and displays the customer names.
   - The programs terminate.

You can rerun the application. In the C program, type 2 to select writing a named pipe, then type a customer number. In the ABL program, select `Read from Pipe`. The C program writes the customer number to the named pipe. The ABL program reads the named pipe, retrieves the row of the customer table with the specified customer number, and displays the row's columns.

14

Shared Library and DLL Support

A *shared library* is a file that contains a collection of compiled functions (routines) that can be accessed by applications. Such a file is called a *shared object* or *shared library* on UNIX and a *dynamic link library* (DLL) in Windows.

An application links to these routines at run time rather than at build time, and shares the code with other applications that link to them. Thus, shared libraries promote code reuse (because an application can reference third-party routines) and facilitate code upgrades (because any enhancement to a shared library becomes immediately available to your application, without rebuilding).

OpenEdge lets you link and execute shared library routines from an ABL procedure. Using these routines, you can write OpenEdge applications that perform a wide range of third-party functions from graphics to advanced multi-media (sound and video) production. You can program how long shared libraries remain memory-resident during application execution.

For more information on shared library concepts and facilities, see the documentation for your operating system.

For details, see the following topics:

- Using shared libraries
- Accessing a shared library entry point
- Passing parameters to a shared library routine
- Executing a shared library routine
- Loading and unloading shared libraries
- Code samples

Using shared libraries

ABL allows you to access shared libraries either statically or dynamically. Static syntax defines the name, signature (parameters), and location of a shared library during development. Dynamic syntax allows for these definitions to be determined at run time.

Consider the following to determine whether static or dynamic access to a shared library is appropriate for your application:

- Static access is appropriate when the number of parameters and their data type is known at compile time
- Dynamic access is appropriate when:
  - The number of parameters and their data type is only known at run time
  - You need to invoke a routine that exists in both a Windows DLL and a UNIX shared library
  - You need to invoke a routine with a variable number of parameters

To access a shared library routine in ABL do the following:

- Declare each shared library routine
  - For static access declare the shared library routine in a manner similar to an ABL internal procedure, using the `PROCEDURE` statement.
  - For dynamic access use the methods and attributes of the call object handle. Create a call object with the `CREATE CALL` statement, then declare each routine using the call object's `CALL-NAME` attribute.
- For each shared library input and output parameter definition, specify the parameter data type that matches the C data type of the corresponding routine parameter
  - For static access, parameters are defined with the `DEFINE PARAMETER` statement
  - For dynamic access, parameters are defined with the call object's `SET-PARAMETER( )` method
- For each ABL variable passed as a routine parameter, ensure that the data type of the variable is compatible with the parameter data type
- Execute the shared library routine
  - For static access, use the `RUN` statement, passing the specified fields, variables, and any required Windows widget handles
  - For dynamic access, use the call object's INVOKE( ) method
- Build and manage any structures used by the routine

The following table compares the ABL elements for declaring, accessing, and executing a shared library routine, both statically and dynamically. Options are shown in brackets.

Table 51: Comparison of static and dynamic shared library access

Task	Static	Dynamic (Call object handle)
Name the access point or shared library routine	`PROCEDURE` *`proc-name`*`:`	`CALL-NAME = "`*`proc-name`*`"`

Task	Static	Dynamic (Call object handle)
Specify the library name	`EXTERNAL "dllname"`	`LIBRARY = "dllname"`
Specify the calling convention	`[ CDECL | PASCAL | STDCALL ]`	`LIBRARY-CALLING-CONVENTION = {"CDECL" | "STDCALL"}`
Specify the number of the entry point for the routine	`[ ORDINAL n ]`	`[ ORDINAL = n ]`
Indicate if the shared library remains in memory	`[ PERSISTENT ]`	`[ PERSISTENT = {TRUE | FALSE }]`
Define the parameters	`[ DEFINE iomode PARAMETER parameterparam-type .]`	`[ SET-PARAMETER ( parameter-number, data-type, iomode, parameter-value) ]`
Parameter *iomode* options	`[ INPUT | OUTPUT | INPUT-OUTPUT | RETURN ]`	`[ INPUT | OUTPUT | INPUT-OUTPUT ]`
Parameter *param-type* options	`[ { LIKE field } | { AS [ HANDLE TO ] data-type } ]`	`[ [ HANDLE TO ] data-type ]`
Expected return data-type by shared library routine	`[ DEFINE RETURN PARAMETER parameter AS data-type ]`	`[ RETURN-VALUE-DLL-TYPE = "data-type" ]`
Execute the library routine	`RUN proc-name [ ( parameter-list) ].`	`INVOKE( )`
Unload the shared library	`RELEASE EXTERNAL [ PROCEDURE ] "dllname".`	`RELEASE EXTERNAL "dllname".`

You can use the call object:

- To invoke an internal or external procedure whose *calling sequence* (number of parameters and the data type of each) is unknown at compile time

  Note: If the name of the procedure is unknown at compile time, use the `RUN` statement with the `VALUE` option, and avoid using the call object.

- To invoke a function whose calling sequence is unknown at compile time

  Note: If the name of the function is unknown at compile time, use the `DYNAMIC-FUNCTION( )` function, and avoid using the call object.

- To reference a widget attribute or a method whose name is unknown at compile time.

  If you already know the name of the attribute or procedure, then you also know its syntax, since the name implies certain syntax. If you know the syntax, then you know the calling sequence, since the syntax defines the calling sequence. If you know the calling sequence, you can use widget:attribute or widget:method syntax, and avoid using the call object.

- To dynamically invoke a shared library routine when:
  - The number of parameters and their data type is only known at run time
  - You need to invoke a routine that exists in both a Windows DLL and a UNIX shared library
  - You need to invoke a routine with a variable number of parameters

Note: The 64-bit Windows GUI and character clients ignore the CDECL, PASCAL, and STDCALL calling conventions if they are specified. The 64-bit Windows GUI and character clients always use the standard 64-bit FASTCALL calling convention.

Accessing a shared library entry point

Depending on how your application accesses a shared library routine, whether statically or dynamically, the ABL elements involved are quite different. For more information about determining which access mode is appropriate for your application, see Using shared libraries on page 484

This section describes the options for defining both static and dynamic access to a shared library.

Declaring a shared library for static access

To statically access a shared library entry point from within ABL, declare the routine using a PROCEDURE statement. The declaration is similar to an internal procedure declaration, but instead of containing procedure code, it contains options and parameter definitions that specify how to access the external shared library routine. This is the syntax for a shared library routine declaration:

Syntax

```
PROCEDURE proc-name  EXTERNAL "dllname"
  [ CDECL | PASCAL | STDCALL ]
  [ ORDINAL n ]
  [ PERSISTENT ] :
  [parameter-definition]...
END [ PROCEDURE ] .
```

Note:

The PROCEDURE statement that declares the DLL routine can appear anywhere within the ABL source file. It does **not** have to precede the RUN statement that invokes it. Typically, the procedure declarations appear at the end of the source file or are written in a separate file that is included in the source file.

The 64-bit Windows GUI and character clients ignore the CDECL, PASCAL, and STDCALL calling conventions if they are specified. The 64-bit Windows GUI and character clients always use the standard 64-bit FASTCALL calling convention.

Options for static access to a shared library routine

In the syntax, the *proc-name* value is the name of your shared library routine. The EXTERNAL option indicates that the procedure being declared is an internal procedure implemented by an external shared library routine. The *dllname* value is the name of the shared library file that contains the routine.

If you specify the PERSISTENT option, the entire shared library remains loaded until ABL exits, or until you explicitly unload the shared library by using the ABL RELEASE EXTERNAL statement.

Each *parameter-definition* value consists of a DEFINE PARAMETER statement. This is the syntax you must use for the DEFINE PARAMETER statement in a DLL routine declaration:

Syntax

```
DEFINE { INPUT | OUTPUT | INPUT-OUTPUT | RETURN }
  PARAMETER parameter
  {{ LIKE field}|{ AS [ HANDLE TO ]data-type}}
```

Each DEFINE PARAMETER statement specifies an INPUT, OUTPUT, or INPUT-OUTPUT parameter in the order it appears in the calling sequence of your shared library routine. You can also (optionally) define one RETURN parameter that provides the function return value of your shared library routine.

The parameter name (*parameter*) serves only as a place holder, and can take any unique identifier value. The AS*date-type* and LIKE*field* options require a special set of data types for shared library parameter definitions.

Windows DLL calling conventions for static access

The PROCEDURE statement allows you to specify the calling convention—C, Pascal, or standard—that your DLL requires. The default is the standard calling convention. Many Windows functions use the standard calling convention. Windows functions that take a variable number of arguments, such as wsprintf( ), often use the C calling convention. For more information on the C, Pascal, and standard calling conventions, see *Microsoft C Language Reference*. For more information on the calling convention that a particular Windows function requires, see *Microsoft Windows Programmer's Reference*. The following table shows how to specify calling conventions.

Table 52: DLL calling conventions

To specify this calling convention...	Code this option...
C	C
Pascal	PASCAL
Standard	STDCALL

You can alternatively specify the DLL entry point by number with the ORDINAL*n* option, where *n* is the ordinal number of the entry point in the library. If you specify the entry point by number, *proc-name* can have any unreserved identifier value, but you must use the same value for the RUN statement that executes the DLL routine.

Note: The 64-bit Windows GUI and character clients ignore the CDECL, PASCAL, and STDCALL calling conventions if they are specified. The 64-bit Windows GUI and character clients always use the standard 64-bit FASTCALL calling convention.

Declaring a shared library for dynamic access

To dynamically access a shared library routine from within ABL, first create a call object with the CREATE CALL statement, which stores the handle to the new call object in the specified variable of type HANDLE. To create a call object, use the following syntax:

Syntax

```
CREATE CALL object-handle [ IN widget-pool ].
```

For example:

```
DEFINE VARIABLE hCall AS HANDLE NO-UNDO.

CREATE CALL hCall.
```

Compared to accessing a shared library statically with the PROCEDURE and DEFINE PARAMETER statements, a shared library is accessed dynamically by setting several call object handle attributes. In the following code fragment, the call object hCall invokes the GetVersion routine in the kernel32.dll shared library:

```
ASSIGN
  hCall:CALL-NAME             = "GetVersion"
  hCall:LIBRARY               = "kernel32.dll"
  hCall:CALL-TYPE             = DLL-CALL-TYPE
  hCall:RETURN-VALUE-DLL-TYPE = "LONG".
```

The CALL-TYPE attribute identifies that the routine exists in a shared library. The routine expects the LONG data type, as specified by the RETURN-VALUE-DLL-TYPE attribute.

Options for dynamic access to a shared library routine

If the PERSISTENT attribute is specified, the entire shared library remains loaded until ABL exits, or until you explicitly unload the shared library by using the ABL RELEASE EXTERNAL statement. For example:

```
hCall:PERSISTENT = TRUE.
```

Use the LIBRARY attribute to specify the shared library name. For example:

```
hCall:LIBRARY = "kernel32.dll".
```

You can use the LIBRARY-CALLING-CONVENTION attribute to specify how you want the shared library to be called. The following character strings are valid LIBRARY-CALLING-CONVENTION values:

- **"CDECL"** — Use the C calling convention
- **"STDCALL"** — Use the standard Windows calling convention

If a value is not specified, `LIBRARY-CALLING-CONVENTION` defaults to "STDCALL". For example:

```
hCall:LIBRARY-CALLING-CONVENTION = "CDECL".
```

The optional `ORDINAL` attribute specifies the number of the DLL entry point to invoke. For UNIX shared libraries this attribute does not apply and is ignored if specified. For example:

```
hCall:ORDINAL = 2.
```

Use the optional `SET-PARAMETER( )` method to define the parameter values, one at a time, to be passed to the shared library routine. This is the syntax for the `SET-PARAMETER( )` method:

Syntax

```
SET-PARAMETER(parameter-number, data-type, iomode, parameter-value)
```

`parameter-number`

An INTEGER expression indicating the order of the parameter. Use 1 for the first parameter, 2 for the second parameter, and so on.

`data-type`

A CHARACTER expression indicating the data type of the parameter. For each parameter, the data type specified by the caller and the callee must be compatible.

When invoking a Windows DLL or a UNIX shared library function, ABL DLL data types are valid as *`data-type`* values. *`data-type`* specifies the type expected by the DLL or the shared library routine parameter. For example, the parameter *`data-type`* is set to "LONG" if the DLL routine parameter expects "LONG". For a list of valid ABL DLL data types, see *ABL Reference*.

`iomode`

A CHARACTER expression indicating the mode of the parameter.

`parameter-value`

An expression whose type is compatible with *`data-type`*. The *`parameter-value`* argument can represent a determinate or indeterminate array. For more information about compatibility of shared library routine and ABL parameter data types, see Table 54: Compatibility of shared library and ABL parameters on page 500.

In the following example, the `NUM-PARAMETERS` attribute is set to the number of parameters that the shared library routine expects, followed by a `SET-PARAMETER( )` method call for each parameter:

```
hCall:NUM-PARAMETERS = 2.
hCall:SET-PARAMETER( 1, "LONG", "INPUT", msecs).
hCall:SET-PARAMETER( 2, "INTEGER", "INPUT", 5000).
```

Passing parameters to a shared library routine

The following sections describe how to pass parameters to a shared library routine.

- Shared library parameter data types on page 490
- Passing arrays as parameters on page 492
- Passing the Unknown value and NULL values on page 496
- Using structure parameters on page 496
- Using MEMPTR variables as parameters on page 497
- DLL routines and ABL widgets (Windows only) on page 498

Shared library parameter data types

For shared library parameter definitions, ABL provides a special set of data types to match the standard C data types used in shared library calling sequences. These are the only data types you can specify for the AS *date-type* option in shared library parameter definitions:

- BYTE
- CHARACTER
- DOUBLE
- FLOAT
- INT64
- LONG
- MEMPTR
- SHORT
- UNSIGNED-LONG
- UNSIGNED-SHORT

CHARACTER and MEMPTR are standard ABL variable data types available for other uses. You can only use the remaining listed data types for shared library parameter definitions.

The MEMPTR data type specifies a pointer to a region of memory. It lets you define and pass C-compatible structures to shared library routines. For more information on using the MEMPTR data type, see Using structure parameters on page 496.

Data type compatibilities

For each parameter definition, you must specify a data type that is compatible with the standard C data type of the corresponding DLL routine parameter. Many data types referenced by DLL routines have the same memory size and usage.

The following table lists each supported memory size and usage, examples of corresponding C data types, and the ABL DLL parameter data type you must use for each one.

Table 53: C and DLL parameter data type compatibilities

Example C data type	DLL parameter data type	Data type size and usage
`char`	`BYTE`	8-bit unsigned integer
`short`	`SHORT`	16-bit signed integer
`unsigned short`	`UNSIGNED-SHORT`	16-bit unsigned integer
`int` (UNIX; Windows) `long` (32-bit UNIX; Windows)	`LONG`	32-bit signed integer
`unsigned int`	`UNSIGNED-LONG`	32-bit unsigned integer
`_int64` (Windows) `long long` (UNIX; Windows) `long` (64-bit UNIX)	`INT64`	64-bit signed integer
`float`	`FLOAT`	4-byte floating point
`double`	`DOUBLE`	8-byte floating point
`char*`	`CHARACTER`	Address (32 bits for 32-bit platforms, 64 bits for 64-bit platforms.)
`c-data-type`[10]	`HANDLE TO` *`parameter-data-type`* [10]	Address (32 bits for 32-bit platforms, 64 bits for 64-bit platforms.)
`char*`, *`output-pointer`* (which can be `char**`, `short**`, and so on), or a pointer to a structure	`MEMPTR`	Address (32 bits for 32-bit platforms, 64 bits for 64-bit platforms.)

To indicate that the DLL or UNIX shared library parameter is a pointer to a value rather than the value itself, use the `HANDLE` option. The `HANDLE` option is required when the DLL routine expects a pointer to the value. Note that the `CHARACTER` data type implies the `HANDLE` option, whether or not you specify it. The `TO` keyword aids readability but has no meaning.

Other data type options

For shared library parameters that pass pointers to scalar values (for example, `SHORT`, `DOUBLE`, etc.), ABL provides the `HANDLE` option. You must use this option for `INPUT` parameters that require pointers to scalar values instead of the values themselves. Although ABL automatically passes pointers for `OUTPUT` and `INPUT-OUTPUT` parameters, the `HANDLE` option is recommended for clarity.

If you use the `LIKE` option to specify the data type of a parameter definition, *`field`* might only be a database field defined as `CHARACTER` or an ABL variable defined as `CHARACTER` or `MEMPTR`.

[10] You can use the HANDLE TO option to specify a pointer to a scalar type. Therefore, you can use the HANDLE TO option with the parameter data types (that is, BYTE, SHORT, UNSIGNED-SHORT, LONG, FLOAT, and DOUBLE) in order to specify a pointer to the respective C data types (that is, char, short, unsigned short, long, int, float, and double). For CHARACTER and MEMPTR parameters, it is redundant because this data type is always passed using a pointer (*`char*)`*.

Note: ABL does not support database fields defined as MEMPTR.

Passing arrays as parameters

This section describes how you can pass arrays as parameters to and from DLL functions.

INTEGER and DECIMAL arrays

When a DLL function requires an array of type BYTE, SHORT, UNSIGNED-SHORT, INT, LONG, FLOAT, or DOUBLE, you can pass an array of the appropriate ABL type (INTEGER or DECIMAL) as a parameter. You can use any parameter mode (INPUT, INPUT-OUTPUT, or OUTPUT). For the data type in the prototype, you can use just the type (for example, LONG) or you can specify it with the HANDLE TO option (for example, HANDLE TO). ABL always passes the array of values as a pointer so the HANDLE TO option is redundant, as it is with any INPUT-OUTPUT or OUTPUT parameter.

Here is an example C Code Prototype for a function called updateCounts:

```
updateCounts(short * parray);
```

This example is the equivalent ABL for the updateCounts procedure when accessing the function statically:

```
DEFINE VARIABLE iCnts AS INTEGER NO-UNDO EXTENT 5.

RUN updateCounts (iCnts).

PROCEDURE updateCounts EXTERNAL "myApp.dll" PERSISTENT:
  DEFINE INPUT-OUTPUT PARAMETER parray AS SHORT.
END.
```

This example is the equivalent ABL for the updateCounts procedure when accessing the function dynamically:

```
DEFINE VARIABLE hCall AS HANDLE  NO-UNDO.
DEFINE VARIABLE ix    AS INTEGER NO-UNDO.
DEFINE VARIABLE mCnts AS MEMPTR  NO-UNDO EXTENT 5.

DO ix = 1 TO EXTENT(mCnts):
  SET-SIZE(mCnts[ix]) = 50.
END.

CREATE CALL hCall.
ASSIGN
  hCall:CALL-NAME             = "updateCounts"
  hCall:LIBRARY               = "myApp.dll"
  hCall:CALL-TYPE             = DLL-CALL-TYPE
  hCall:RETURN-VALUE-DLL-TYPE = "INTEGER"
  hCall:PERSISTENT            = TRUE
  hCall:NUM-PARAMETERS        = 1.

hCall:SET-PARAMETER(1, "SHORT", "INPUT-OUTPUT", mCnts).
hCall:INVOKE( ).
```

INT64 arrays

OpenEdge also supports passing INT64 EXTENT variables (arrays) to DLL entry points, and supporting INT64 arrays in DLL parameters. These cases follow the same conversion rules for scalar INT64 variables. Specifically:

- An EXTENT of INT64 passed to a BYTE, SHORT, UNSIGNED-SHORT, LONG or UNSIGNED-LONG INPUT DLL parameter constructs an array of the appropriate data type, containing the appropriate low-order byte values. If any of the values in the EXTENT exceed the capacity of the target data type, ABL generates an overflow error.
- An EXTENT of INT64 passed to an INT64 INPUT DLL parameter constructs an array of the same length containing 64-bit integer values.
- An EXTENT of INTEGER passed to an INT64 INPUT DLL parameter constructs an array of 64-bit integers, automatically converting the INTEGER values to INT64 values.
- An EXTENT of INT64 passed to a BYTE, SHORT, UNSIGNED-SHORT, LONG or UNSIGNED-LONG INPUT-OUTPUT DLL parameter constructs an array of the appropriate data type, containing the appropriate low-order byte values. If any of the values in the EXTENT exceed the capacity of the target data type, ABL generates an overflow error at runtime. ABL automatically converts the returned values to INT64.
- An EXTENT of INT64 passed to an INT64 INPUT-OUTPUT DLL parameter constructs an array containing 64-bit integer values.
- An EXTENT of INTEGER passed to an INT64 INPUT-OUTPUT DLL parameter constructs an array of 64-bit integer values, automatically converting the INTEGER values to 64-bit values. ABL automatically converts the returned values from INT64 to INTEGER, removing the high-order bytes where necessary. If any of the extent values are larger than what would fit in an INTEGER, the ABL client generates an overflow error.
- An EXTENT of INTEGER passed to an UNSIGNED-LONG INPUT-OUTPUT DLL parameter constructs an array of 32-bit unsigned integer values. ABL automatically converts the returned values from UNSIGNED-LONG to INTEGER. If any of the extent values are larger than what would fit in an INTEGER, the ABL client generates an overflow error at runtime.
- An EXTENT of INT64 passed to a BYTE, SHORT, UNSIGNED-SHORT, LONG or UNSIGNED-LONG OUTPUT DLL parameter constructs an array of the same length for the appropriate data type. ABL automatically converts the returned values to INT64.
- An EXTENT of INT64 passed to an INT64 OUTPUT DLL parameter constructs an array of 64-bit integers.
- An EXTENT of INTEGER passed to an INT64 OUTPUT DLL parameter constructs an array of 64-bit integers. ABL automatically converts the returned values from INT64 to INTEGER, removing the high-order bytes where necessary. If any of the extent values are larger than what would fit in an INTEGER, the ABL client generates an overflow error at runtime.
- An EXTENT of INTEGER passed to an UNSIGNED-LONG OUTPUT DLL parameter constructs an array of 32-bit unsigned integer values. ABL automatically converts the returned values from UNSIGNED-LONG to INTEGER. If any of the EXTENT values are larger than what would fit in an INTEGER, the ABL client generates an overflow error at runtime.

Note: You **cannot** pass an array of indeterminate extent to a DLL.

CHARACTER and LONGCHAR arrays

If the DLL requires an array of strings as INPUT which is represented by an array of pointers to strings, you can pass an ABL CHARACTER or LONGCHAR array. You can do the same if it is an INPUT-OUTPUT parameter, however, this is **not** recommended. If the DLL updates the data such that the output string is longer than the input string, this can result in a memory exception or other unpredictable behavior. Therefore, for INPUT-OUTPUT parameters you should use a MEMPTR array, described in MEMPTR arrays on page 495.

Also, just as you cannot pass CHARACTER or LONGCHAR OUTPUT parameters to a DLL, you also cannot pass a CHARACTER or LONGCHAR array as an OUTPUT parameter. If the DLL requires an array of character buffers that it will modify or if the DLL allocates memory for a set of strings and returns them as an array of pointers, you should use a MEMPTR parameter.

As with the INTEGER and DECIMAL types of arrays, the data type in the prototype can be either just the type (for example, CHARACTER) or you can specify it with the HANDLE TO option.

Here is an example C Code Prototype for a function called nameLookup:

```
nameLookup(char **ppStrArray, char *pfindName);
```

This example is the equivalent ABL for the nameLookup procedure when accessing the function statically:

```
DEFINE VARIABLE cFindName AS CHARACTER NO-UNDO.
DEFINE VARIABLE cNames    AS CHARACTER NO-UNDO EXTENT 50.

RUN nameLookup (cNames, cFindName).

PROCEDURE nameLookup EXTERNAL "myApp.dll" PERSISTENT:
  DEFINE INPUT PARAMETER ppStrArray AS HANDLE TO CHARACTER.
  DEFINE INPUT PARAMETER pfindName  AS CHARACTER.
END.
```

This example is the equivalent ABL for the nameLookup procedure when accessing the function dynamically:

```
DEFINE VARIABLE cFindName AS CHARACTER NO-UNDO.
DEFINE VARIABLE cNames    AS CHARACTER NO-UNDO EXTENT 50.
DEFINE VARIABLE hCall     AS HANDLE    NO-UNDO.

CREATE CALL hCall.
ASSIGN
  hCall:CALL-NAME             = "nameLookup"
  hCall:LIBRARY               = "myApp.dll"
  hCall:CALL-TYPE             = DLL-CALL-TYPE
  hCall:RETURN-VALUE-DLL-TYPE = "CHARACTER"
  hCall:PERSISTENT            = TRUE
  hCall:NUM-PARAMETERS        = 2.

hCall:SET-PARAMETER(1, "CHARACTER", "INPUT", cNames).
hCall:SET-PARAMETER(2, "CHARACTER", "INPUT", cFindName).
hCall:INVOKE( ).
```

MEMPTR arrays

If the DLL requires an array of pointers, where each pointer either points at a string or a structure, you can pass an ABL MEMPTR array. For INPUT or INPUT-OUTPUT parameters, you initialize each MEMPTR as appropriate to contain the string or structure to pass for INPUT. If it is an INPUT-OUTPUT parameter, you must insure that the allocated size (that is, the size set by SET-SIZE) is large enough to hold any expected output value, not just large enough to hold the input value. For an OUTPUT parameter, if the DLL expects an array of pointers to allocated buffers, you must use SET-SIZE to the required size for each MEMPTR in the array.

If the DLL allocates memory for a set of strings or structures and returns them as an array of pointers, you must set each MEMPTR to at least the size of a pointer (for example, 4 or 8 bytes depending on the platform). You then use the GET-POINTER-VALUE and SET-POINTER-VALUE functions to access the data on return. (For more information on the GET- and SET-POINTER-VALUE functions, see *ABL Reference*.) If you want to pass NULL as one or more of the pointers, the corresponding MEMPTR in the array can be a *null MEMPTR*, meaning that it is either uninitialized or that you did a SET-SIZE on it to set it to zero length. For information on passing a MEMPTR parameter, see Using MEMPTR variables as parameters on page 497.

Here is an example C Code Prototype for a function called getNames:

```
getNames(char **pnameList);
```

This example is the equivalent ABL for the getNames procedure when accessing the shared library statically. In this case, the DLL does not allocate the buffer, but expects to be passed an array of buffer pointers:

```
DEFINE VARIABLE ix     AS INTEGER NO-UNDO.
DEFINE VARIABLE mNames AS MEMPTR  NO-UNDO EXTENT 10.

DO ix = 1 TO EXTENT(mNames):
  SET-SIZE(mNames[ix]) = 50.
END.

RUN getNames (mNames).

DO ix = 1 TO EXTENT(mNames):
  DISPLAY GET-STRING(mNames[ix], 1).
END.

PROCEDURE getNames EXTERNAL "myApp.dll" PERSISTENT:
  DEFINE OUTPUT PARAMETER names AS MEMPTR NO-UNDO.
END.
```

This example is the equivalent ABL for the `getNames` procedure when accessing the shared library dynamically:

```
DEFINE VARIABLE hCall  AS HANDLE  NO-UNDO.
DEFINE VARIABLE ix     AS INTEGER NO-UNDO.
DEFINE VARIABLE mNames AS MEMPTR  NO-UNDO EXTENT 10.

DO ix = 1 TO EXTENT(mNames):
  SET-SIZE(mNames[ix]) = 50.
END.

CREATE CALL hCall.
ASSIGN
  hCall:CALL-NAME             = "getNames"
  hCall:LIBRARY               = "myApp.dll"
  hCall:CALL-TYPE             = DLL-CALL-TYPE
  hCall:RETURN-VALUE-DLL-TYPE = "CHARACTER"
  hCall:PERSISTENT            = TRUE
  hCall:NUM-PARAMETERS        = 1.

hCall:SET-PARAMETER(1, "MEMPTR", "INPUT", mNames).
hCall:INVOKE( ).

DO ix = 1 TO EXTENT(mNames):
  DISPLAY GET-STRING(mNames[ix], 1).
END.
```

Passing the Unknown value and NULL values

If you have to pass a `NULL` value to a shared library routine, do not pass a null `MEMPTR` variable.

Instead, define the `INPUT` or `INPUT-OUTPUT` parameter as a `LONG` (32-bit platforms) or `INT64` (64-bit platforms), and pass it a zero value (0) when you run the routine. If this conflicts with calling the shared library entry point another way, you can create a second declaration using the `ORDINAL` option of the `PROCEDURE` statement. (Note that this option is not available on UNIX.)

Using this technique does not conform to best practices with respect to portability because you must know whether the platform is 32-bit or 64-bit in order to declare the entry point correctly (that is, with a `LONG` or `INT64`) and to call it correctly. However, since using DLLs is also not considered best practice for portability, using this technique is a less serious compromise to the goal of platform-independence for applications.

One exception to the rule of not passing a null `MEMPTR` variable is a `MEMPTR` array, in which you can pass a `MEMPTR` of size 0.

You **cannot** pass the Unknown value (?), regardless of the parameter's data type.

Using structure parameters

Many shared library routines require a pointer to a structure rather then a scalar value. ABL provides the `DEFINE VARIABLE...AS MEMPTR` or `VAR MEMPTR...` statement to define the variable. Once defined, you can use the `MEMPTR` variable with a set of ABL statements and functions to build, read, and modify the contents of any C-compatible structure used by your shared library routines. For basic information on the `MEMPTR` data type and how to use it, see Introduction to External Program Interfaces on page 413. This section provides information on using `MEMPTR` variables as parameters to shared library routines.

Note: You **cannot** use the ABL `RAW` data type to build structures that you pass to shared library routines.

Using MEMPTR variables as parameters

This section describes how to set the memory size for MEMPTR variables and how to pass a character string in a MEMPTR memory region.

Initializing and uninitializing MEMPTR variables

Typically when you initialize a MEMPTR variable, you use the SET-SIZE statement to allocate a region of memory for a specified size and associate it with the variable. You can also use a shared library routine to allocate the memory for a MEMPTR variable by passing the variable appropriately as a RETURN parameter to the routine. Then, to complete MEMPTR initialization, you should use the SET-SIZE statement so that ABL knows how big the memory area is.

You **must** know the exact size of data returned by the shared library routine to initialize the MEMPTR variable properly with the SET-SIZE statement. If you use an incorrect value, you might not be able to access the data as you expect. Note also that if you do not complete initialization of a shared library pre-initialized MEMPTR variable using the SET-SIZE statement, ABL does not perform any bounds checking when you read or modify contents of the structure.

Caution: If you specify a size that is too small, ABL prevents you from accessing parts of the returned structure that lie outside the specified region. However, if you specify a size that is too large (or do not complete initialization at all), you might cause a memory violation by inappropriately accessing memory outside the area of the structure. This can result in loss of data. To determine the size of structures allocated and returned by shared library routines, see the documentation for the routine you are calling.

Passing CHARACTER values to shared library routines

If you are passing an ABL character string to a shared library routine, you can pass it as a CHARACTER variable or expression. However, if you expect the shared library routine to modify the value, Progress Software Corporation recommends that you pass the ABL character string in a MEMPTR memory region as a null-terminated string. This is required if it is an OUTPUT parameter. Otherwise, the shared library routine might inappropriately modify ABL memory outside the bounds of the CHARACTER value with unpredictable results. Assuming ABL has allocated the memory for the string (using SET-SIZE), you then use the GET-STRING function to extract the CHARACTER value.

If the DLL allocates the memory as either a RETURN value or an OUTPUT parameter, you must use an OUTPUT MEMPTR. For a RETURN parameter, you can use GET-STRING (or another access function) directly on the MEMPTR on return. However, for an OUTPUT parameter, you must use the GET-POINTER-VALUE and SET-POINTER-VALUE functions to access the data on return. (For more information on the GET- and SET-POINTER-VALUE functions, see *ABL Reference*.)

Freeing memory associated with a MEMPTR variable

The region of memory associated with a MEMPTR variable remains allocated until it is freed. In some cases, the shared library routine frees the memory; in other cases, the calling procedure must free the memory using the SET-SIZE statement to set its size to zero (0). ABL cannot free the memory for you. It is up to you to ensure that the memory is freed, depending on the functionality of each shared library routine you use.

The following code fragment shows how to pass an INPUT-OUTPUT string value as a MEMPTR parameter in the case where ABL allocates the memory:

```
DEFINE VARIABLE cString AS CHARACTER NO-UNDO.
DEFINE VARIABLE iLength AS INTEGER   NO-UNDO.
DEFINE VARIABLE iSize   AS INTEGER   NO-UNDO.
DEFINE VARIABLE mVar    AS MEMPTR    NO-UNDO.

/* The longest string the DLL returns is 256 characters */
ASSIGN
  iLength            = LENGTH(cString)
  iSize              = (IF iLength > 256 THEN iLength ELSE 256)
  SET-SIZE(mVar)     = iSize
  PUT-STRING(mVar,1) = cString.

RUN DLLfunction (INPUT-OUTPUT mVar).

cString = GET-STRING(mVar, 1).

PROCEDURE DLLfunction EXTERNAL "anysystem.dll" ORDINAL 10:
  DEFINE INPUT-OUTPUT PARAMETER StringParm AS MEMPTR NO-UNDO.
END PROCEDURE.
```

The DLL routine is the tenth function in the anysystem.dll file. The SET-SIZE statement allocates to mVar a memory region large enough to hold both the input and output cString values. The PUT-STRING statement stores the cString value in mVar. After passing mVar to the DLL routine, the GET-STRING statement returns the (new) value to cString.

DLL routines and ABL widgets (Windows only)

Some system DLL routines can manipulate windows on the display. To allow these routines to interact with ABL windows, every ABL user interface widget has the HWND attribute. This attribute contains the Windows handle to the window that contains the widget. You can pass this window handle as an integer value to a DLL routine using a LONG DLL parameter or a LONG location in a structure (MEMPTR) parameter.

Caution: When you pass an HWND to a DLL routine, that routine has complete control of a system window that ABL is using for its own widget management. This means that the DLL routine can modify the window's attributes and even destroy the window itself without ABL knowing about it. Such actions can cause unintended effects on your OpenEdge application.

Executing a shared library routine

The following sections describe how you invoke a shared library routine that has been accessed statically as compared to dynamically.

- Executing a shared library routine for static access on page 499
- Executing a shared library routine for dynamic access on page 499
- Parameter data types on page 499
- INT64 parameter conversions on page 500

Executing a shared library routine for static access

Use the ABL RUN statement to execute shared library routines that have been accessed statically. This is the syntax for RUN statements that execute shared library routines:

Syntax

```
RUN proc-name [ ( parameter-list ) ]
```

You use this statement in virtually the same way as for internal ABL procedures or external procedure files. The only differences are that positional arguments referenced in procedure files by ordinal identifiers ({1}, {2}, etc.) have no meaning for shared library routines (which cannot compile them), and errors returned by shared library routines do not raise the ABL error condition. They might, however, return error messages.

Options for shared library routine execution

The *proc-name* in the RUN statement must match the *proc-name* in the corresponding PROCEDURE statement, whether or not you declare the shared library routine with the ORDINAL option. Also, *proc-name* must be declared in the same ABL source file where the RUN statement is executed.

The *parameter-list* contains INPUT, OUTPUT, and INPUT-OUTPUT parameters in the order defined for the corresponding PROCEDURE statement. If you specify a RETURN parameter for the PROCEDURE statement, you must match it with an OUTPUT parameter in the corresponding RUN statement. You cannot specify a RETURN parameter for a shared library function as a CHARACTER variable. You must use a MEMPTR variable to return a character string. See Passing CHARACTER values to shared library routines on page 497 for more information. The RUN statement parameter data types must match the data types of the corresponding parameter definitions for the PROCEDURE statement. Any mismatch causes a run-time error.

Executing a shared library routine for dynamic access

Use the INVOKE( ) method on the call object handle to execute a shared library routine that has been dynamically accessed. This is the syntax for INVOKE( ) method:

Syntax

```
call-object-handle:INVOKE( ).
```

Parameter data types

While shared library parameter definitions use a special set of data types, the corresponding expressions, fields, and variables passed in the RUN statement or SET-PARAMETER( ) method of the call object handle have standard ABL data types. You must ensure that your ABL parameters have data types that are compatible with their corresponding shared library parameter definitions. The following table lists each shared library parameter data type and its compatible ABL data type.

Table 54: Compatibility of shared library and ABL parameters

Shared library parameter data types	ABL parameter data types
BYTE	INTEGER, INT64
SHORT	INTEGER, INT64
UNSIGNED-SHORT	INTEGER, INT64
LONG	INTEGER, INT64
UNSIGNED-LONG	INTEGER, INT64
INT64	INTEGER, INT64
FLOAT	DECIMAL
DOUBLE	DECIMAL
CHARACTER	CHARACTER
MEMPTR	MEMPTR

Caution: For CHARACTER parameters, ABL always passes the routine a pointer to the character or character string value rather than the value itself. If the routine modifies the value, it can also modify ABL memory outside the bounds of the CHARACTER value with unpredictable results. For this reason, ABL does not allow you to use OUTPUT or RETURN for CHARACTER or LONGCHAR parameters, as well as CHARACTER or LONGCHAR array parameters. PSC does not recommend using INPUT-OUTPUT for CHARACTER or LONGCHAR parameters. Rather, pass the character string as a MEMPTR parameter. For more information, see Passing CHARACTER values to shared library routines on page 497. For more information on passing a MEMPTR parameter, see Using MEMPTR variables as parameters on page 497.

Note: You cannot use any type of array for a RETURN parameter.

INT64 parameter conversions

Differing integer sizes for DLL parameter data types and ABL data types require automatic conversions between these types. Passing a larger data type to a smaller data type will convert the larger data type to the smaller data type. If the value of the larger data type is greater than what would fit in the smaller data type, ABL generates an overflow error at run time. Passing a smaller data type to a larger data type automatically converts the smaller data type to a larger data type.

For example:

- When passing an INT64 value to a BYTE, SHORT, UNSIGNED-SHORT, LONG or UNSIGNED-LONG DLL parameter, ABL automatically passes the appropriate number of low-order bytes of the INT64 value to the DLL function. If the INT64 value is larger than what would fit in the target data type, ABL generates an overflow error.
- When passing an INTEGER value to an INT64 DLL parameter, ABL provides an automatic conversion of the value to INT64.

- When returning a BYTE, SHORT, UNSIGNED-SHORT, LONG or UNSIGNED-LONG INPUT-OUTPUT or OUTPUT DLL parameter to an INT64 variable, ABL provides an automatic conversion to INT64.
- When returning a BYTE, SHORT, UNSIGNED-SHORT, LONG or UNSIGNED-LONG RETURN DLL parameter to an INT64 variable, ABL provides an automatic conversion to INT64.
- When returning an INT64 DLL parameter to an INTEGER variable, ABL passes only the low-order 4 bytes to the INTEGER, ignoring the high-order bytes. If the returned value is larger than what would fit in an INTEGER, ABL client generates an overflow error.
- When returning an UNSIGNED-LONG to an INTEGER variable, ABL provides an automatic conversion to INTEGER. If the UNSIGNED-LONG value is larger than the maximum value of an INTEGER, ABL generates an overflow error.

Loading and unloading shared libraries

ABL provides several options for loading and unloading shared libraries. How this is done in ABL depends on how you access the shared library, either statically or dynamically. The following table compares these options for statically and dynamically accessed shared libraries.

Table 55: Loading and unloading shared libraries

If you want to ...	Static	Dynamic(Call object handle)
Load and unload a shared library each time you invoke it	PROCEDURE statement without the PERSISTENT option	*call-object*:PERSISTENT = FALSE or do not specify the attribute
Load a shared library and keep it loaded	PROCEDURE statement with the PERSISTENT option	*call-object*:PERSISTENT = TRUE
Manually unload a shared library	RELEASE EXTERNAL statement	RELEASE EXTERNAL statement

Code samples

The following code samples provide examples of working with shared library routines, including defining access, setting parameters, setting attributes, and invoking routines.

- Examples of static access to a shared library on page 501
- Examples of dynamic access to a shared library on page 504

Examples of static access to a shared library

The following examples show several variations on the use of the PROCEDURE and DEFINE PARAMETER statements for static access to a shared library.

In this example, the ABL MESSAGE statement displays a message in an alert box. It then calls the MessageBoxA routine from user32.dll to display the same message in an identical alert box, as shown:

i-dllex1.p

```
/* i-dllex1.p */

DEFINE VARIABLE iResult AS INTEGER NO-UNDO.

MESSAGE "   It's a whole new world!"
  VIEW-AS ALERT-BOX MESSAGE BUTTONS OK TITLE "ABL DLL Access".

RUN MessageBoxA (0, "   It's A Whole New World!",
  "ABL DLL Access - from the DLL!", 0, OUTPUT iResult).

PROCEDURE MessageBoxA EXTERNAL "user32.dll":
  DEFINE INPUT PARAMETER hwnd    AS LONG.
  DEFINE INPUT PARAMETER mbtext  AS CHARACTER.
  DEFINE INPUT PARAMETER mbtitle AS CHARACTER.
  DEFINE INPUT PARAMETER style   AS LONG.
  DEFINE RETURN PARAMETER result AS LONG.
END.
```

The following procedure uses the `sndPlaySoundA` routine from `winmm.dll`. The procedure allows the user to select a sound to play and then invokes the DLL routine to play the sound. The DLL routine takes two input parameters and returns a status code, as shown:

i-dllex2.p

```
DEFINE VARIABLE wave-name   AS CHARACTER NO-UNDO INITIAL ?.
DEFINE VARIABLE play-status AS INTEGER   NO-UNDO.

SYSTEM-DIALOG GET-FILE wave-name
  TITLE "Choose the Sound"
  FILTERS "Wave Files (*.wav)" "*.wav"
  MUST-EXIST USE-FILENAME.

RUN sndPlaySoundA (INPUT wave-name, INPUT 2, OUTPUT play-status).

PROCEDURE sndPlaySoundA EXTERNAL "winmm.dll":
  DEFINE INPUT PARAMETER ic   AS CHARACTER.
  DEFINE INPUT PARAMETER ish  AS LONG.
  DEFINE RETURN PARAMETER osh AS LONG.
END PROCEDURE.
```

Note: You must have a sound driver installed on your machine to play sounds.

The following code sample demonstrates calling the C library function `atoi` to get the value of the character form of an integer:

```
DEFINE VARIABLE in-string AS MEMPTR  NO-UNDO.
DEFINE VARIABLE out-int   AS INTEGER NO-UNDO.

ASSIGN
  SET-SIZE(in-string)      = 10
  PUT-STRING(in-string, 1) = "150".

RUN atoi (INPUT in-string, OUTPUT out-int).
MESSAGE "ATOI RESULT: " out-int VIEW-AS ALERT-BOX.

PROCEDURE atoi EXTERNAL "/usr/lib/libc.so.1":
  DEFINE INPUT  PARAMETER b       AS MEMPTR.
  DEFINE OUTPUT PARAMETER ret-val AS SHORT.
END PROCEDURE.
```

The declaration of the C function being called looks like this:

```
int atoi(const char *str)
```

The following procedure defines and displays a shaded ellipse in the current window using DLL functions from the Windows graphics library. This requires initialization of a small structure (`ElipRegion`).

Note that ABL has no knowledge of any graphics that you create using DLLs. You must ensure that ABL does not refresh the window you are using while the graphics are displayed. Otherwise, the graphics disappear during a window system refresh. (You can help to mitigate this by providing a graphics refresh option within your OpenEdge application.) The `i-dllex3.p` procedure displays a preparatory message ("Preparing drawing"), and pauses to realize the current window before calling the DLL routines that display the filled ellipse. The procedure pauses by default before it terminates, allowing the ellipse to remain on the display, as shown:

i-dllex3.p

```
/* DLL routine to create an elliptic region */
PROCEDURE CreateEllipticRgnIndirect EXTERNAL "gdi32.dll":
  DEFINE RETURN PARAMETER RegionHandle AS LONG.
  DEFINE INPUT  PARAMETER RegionSpec   AS MEMPTR.
END PROCEDURE.

/* DLL routine to get drawing object */
PROCEDURE GetStockObject EXTERNAL "gdi32.dll":
  DEFINE RETURN PARAMETER ObjectHandle AS LONG.
  DEFINE INPUT  PARAMETER ObjectType   AS LONG.
END PROCEDURE.

/* DLL routine to select region into device context */
PROCEDURE SelectObject EXTERNAL "gdi32.dll":
  DEFINE INPUT PARAMETER DeviceHandle AS LONG.
  DEFINE INPUT PARAMETER ObjectHandle AS LONG.
END PROCEDURE.

/* DLL routine to display region */
PROCEDURE PaintRgn EXTERNAL "gdi32.dll":
  DEFINE INPUT PARAMETER DeviceHandle AS LONG.
  DEFINE INPUT PARAMETER RegionHandle AS LONG.
END PROCEDURE.

/* DLL routine to get handle of window device context */
PROCEDURE GetDC EXTERNAL "user32.exe":
```

```
  DEFINE RETURN PARAMETER DeviceHandle AS LONG.
  DEFINE INPUT  PARAMETER WindowHandle AS LONG.
END PROCEDURE.

/* DLL routine to release device context handle */
PROCEDURE ReleaseDC EXTERNAL "user32.exe":
  DEFINE INPUT PARAMETER DeviceHandle AS LONG.
END PROCEDURE.

/* DLL routine to delete elliptic region */
PROCEDURE DeleteObject EXTERNAL "gdi32.dll":
  DEFINE INPUT PARAMETER RegionHandle AS LONG.
END PROCEDURE.

/* Variable Definitions */
DEFINE VARIABLE ElipRegion AS MEMPTR  NO-UNDO. /* Elliptic region struct. */
DEFINE VARIABLE hDevice    AS INTEGER NO-UNDO. /* Device context handle */
DEFINE VARIABLE hObject    AS INTEGER NO-UNDO. /* Drawing object handle */
DEFINE VARIABLE hRegion    AS INTEGER NO-UNDO. /* Elliptic region handle */
DEFINE VARIABLE erLeft   AS INTEGER INITIAL 1  NO-UNDO. /* Elliptic */
DEFINE VARIABLE erTop    AS INTEGER INITIAL 5  NO-UNDO. /* Coordinates */
DEFINE VARIABLE erRight  AS INTEGER INITIAL 9  NO-UNDO.
DEFINE VARIABLE erBottom AS INTEGER INITIAL 13 NO-UNDO.

/* Allocate and build elliptic region structure specifying rectangular
   coordinates of region */
SET-SIZE(ElipRegion) = 4  /* int left   */
                     + 4  /* int top    */
                     + 4  /* int right  */
                     + 4. /* int bottom */

PUT-LONG(ElipRegion, erLeft)   = 50.
PUT-LONG(ElipRegion, erTop)    = 50.
PUT-LONG(ElipRegion, erRight)  = 200.
PUT-LONG(ElipRegion, erBottom) = 100.

/* Initialize current window with PAUSE and display perfunctory message */
DISPLAY "Preparing drawing..." .
PAUSE.
/* Get device context, region, and drawing object handles */
RUN GetDC (OUTPUT hDevice, INPUT CURRENT-WINDOW:HWND).
RUN CreateEllipticRgnIndirect (OUTPUT hRegion, INPUT ElipRegion).
RUN GetStockObject (OUTPUT hObject, INPUT 4).

/* Select drawing object and region for device context, and paint region */
RUN SelectObject (INPUT hDevice, INPUT hObject).
RUN SelectObject (INPUT hDevice, INPUT hRegion).
RUN PaintRgn (INPUT hDevice, INPUT hRegion).

/* Free resources */
/* Free region structure */
SET-SIZE(ElipRegion) = 0.

/* Release device context */
RUN ReleaseDC (INPUT CURRENT-WINDOW:HWND, INPUT hDevice).

/* Delete elliptic region */
RUN DeleteObject (INPUT hRegion).

/* Wait for user to close window or hit Escape */
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

Examples of dynamic access to a shared library

The following examples show several variations on the use of the call object handle attributes for dynamic access to a shared library.

This example demonstrates the use of the RETURN-VALUE-DLL-TYPE attribute and the RETURN-VALUE attribute when invoking a Windows DLL routine. RETURN-VALUE-DLL-TYPE is set to "LONG", which is the value that the DLL routine expects to receive, as shown:

```
FUNCTION GetWinVersion RETURNS INTEGER:
  DEFINE VARIABLE iValue  AS INTEGER   NO-UNDO.
  DEFINE VARIABLE libName AS CHARACTER NO-UNDO.
  DEFINE VARIABLE hCall   AS HANDLE    NO-UNDO.

  CREATE CALL hCall.
  ASSIGN
    hCall:CALL-NAME             = "GetVersion"
    hCall:LIBRARY               = "kernel32.dll"
    hCall:CALL-TYPE             = DLL-CALL-TYPE
    hCall:RETURN-VALUE-DLL-TYPE = "LONG".
  hCall:INVOKE( ).
  iValue = hCall:RETURN-VALUE.

  DELETE OBJECT hCall.
  RETURN iValue.
END FUNCTION.
```

After the invoke finishes executing, iValue contains an INTEGER value.

The following example implements an ABL function, sleep, which causes the AVM to sleep for a specified number of milliseconds. The code checks to determine which OS is running, and invokes the appropriate Windows DLL or UNIX shared library.

```
FUNCTION sleep (msecs AS INTEGER):
  DEFINE VARIABLE libName AS CHARACTER NO-UNDO
  DEFINE VARIABLE hCall   AS HANDLE    NO-UNDO.

  libName = IF OPSYS = "WIN32" THEN "kernel32.dll" ELSE "libc.so.1".

  CREATE CALL hCall.
  ASSIGN
    hCall:CALL-NAME      = "sleep"
    hCall:LIBRARY        = libName
    hCall:CALL-TYPE      = DLL-CALL-TYPE
    hCall:NUM-PARAMETERS = 1.

  hCall:SET-PARAMETER(1, "LONG", "INPUT", msecs).
  hCall:INVOKE( ).

  DELETE OBJECT hCall.
  RETURN msecs.
END FUNCTION.
```

15

Windows Dynamic Data Exchange

Dynamic data exchange (DDE) is a general protocol for inter-process communication in Windows.

For details, see the following topics:

- Introduction to DDE
- Using the DDE protocol
- Structuring a DDE conversation in ABL
- Defining conversation endpoints: DDE frames
- Opening DDE conversations
- Exchanging data in conversations
- Closing DDE conversations
- DDE example

Introduction to DDE

The DDE protocol allows any two Windows applications to communicate as client and server, where the client initiates communications with the server and the server provides data and services to the client.

Note: OpenEdge support for DDE is a deprecated feature. OpenEdge supports DDE for backward compatibility only. For inter-process communication on Windows platforms, consider using the Microsoft Component Object Model (COM). OpenEdge support for COM is documented in Using COM Objects in ABL on page 523.

OpenEdge supports DDE communications acting as a DDE client only. Using ABL (Advanced Business Language) DDE statements, your OpenEdge application can initiate and maintain communications with any other Windows application with DDE server capability. This communications includes exchanging data with the server and directing the server to execute server-defined commands. Your OpenEdge application can communicate with different server applications during a session, and can even access entirely new server applications without modifying your existing ABL code.

For more information on DDE concepts and facilities, see the Windows programming documentation on DDE.

Using the DDE protocol

The DDE protocol establishes communications between two applications through a *conversation*. A conversation provides a unique conduit through which they can exchange information. One application is the *DDE server* and the other the *DDE client*. It is the responsibility of the DDE client to open and manage conversations with a DDE server. The DDE server responds to client requests to send or receive information, and to execute server commands on the client's behalf.

A OpenEdge application is always the DDE client and can only open conversations with applications that act as DDE servers. Thus, two OpenEdge clients cannot use DDE to converse with each other (except through an intermediary server).

The course of a DDE conversation

All DDE conversations follow a pattern of execution that varies somewhat with the application.

To open and manage a DDE conversation:

1. Select a named frame to use as a conversation endpoint (*DDE frame*). DDE attributes of this frame maintain the status of each conversation you open with it.

   Note: A DDE frame cannot be a dialog box (no `VIEW-AS DIALOG-BOX` option).

2. Ensure that the DDE server application is running in your Windows environment and that the DDE frame for your application is realized.
3. Open the conversation to the server application, specifying the DDE frame, application, and topic names. A topic is a category defined by the server that includes specific data items or commands that the client can access.
4. If you want to, send commands to the DDE server that define additional topics or otherwise prepare the server to open DDE conversations. (You typically open an initial conversation for the System topic to execute these commands.)
5. Send and receive data values between your OpenEdge client and DDE server using the data items associated with your topic of conversation. You can converse with server data items on demand, or set up links that let your OpenEdge application automatically receive the data when server data items change value.
6. To close a conversation, remove any links established to associated data items and terminate the conversation.

ABL statements for DDE conversations

OpenEdge provides several ABL statements to open and manage DDE conversations. The following table lists these DDE statements.

Table 56: DDE statements

Statement	Description
`DDE ADVISE`	Creates or removes a link to a DDE server data item. Creating a link allows ABL to trigger a `DDE-NOTIFY` event when the data item changes value.
`DDE EXECUTE`	Sends one or more application commands to the DDE server to execute.
`DDE GET`	Retrieves the new value of a linked data item in response to a `DDE-NOTIFY` event for the data item. You typically invoke this command in a DDE frame trigger block for the `DDE-NOTIFY` event.
`DDE INITIATE`	Opens a DDE conversation.
`DDE REQUEST`	Requests the current value of a server data item.
`DDE SEND`	Sends a new value to a server data item.
`DDE TERMINATE`	Closes a DDE conversation.

The following sections provide more information on how ABL structures DDE conversations and how to use these ABL statements to open, manage, and close DDE conversations. For a complete description of each statement, see *ABL Reference*.

Structuring a DDE conversation in ABL

In a DDE conversation, the OpenEdge client sends and receives data values between ABL fields, variables, and expressions at one end and corresponding data items in the server application at the other end. The client can also send commands to the server using whatever language or format that the server recognizes. The documentation for the server application generally tells you how to set up DDE conversations. This information includes how to address the server from any client, such as OpenEdge. In general, ABL structures all DDE conversations within a standard hierarchy.

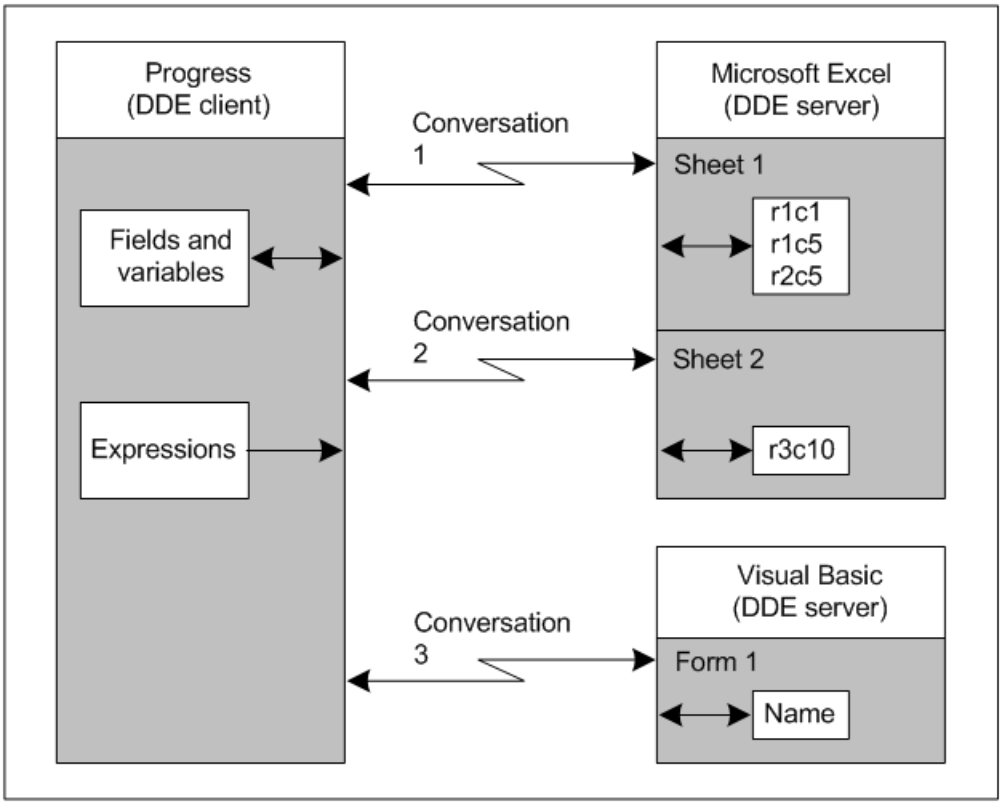
Conversation hierarchy

Any DDE conversation is structured in a hierarchy that consists of an *application name* that identifies the DDE server, a *topic name* within the application, and one or more *data item names* within the topic. The application and topic name together with a DDE frame uniquely identifies the conversation, which ABL also assigns a unique *channel number*.

The application name is a name defined by the server that is unique to other DDE server applications. This is usually the filename of the executable without the extension (for example, EXCEL for Microsoft Excel). The topic name identifies a category used to group server data items. If the data items are in a file accessed by the server, the topic name might be the filename (for example, a worksheet name such as Sheet1 in Excel). The data item names identify the data items that the server defines for the topic. These names generally follow a convention defined by and used in the server application itself (for example, the row and column coordinates of a worksheet cell, such as R3C12 in Excel). Once the OpenEdge client opens a conversation, it might exchange values with any server data item associated with the topic of conversation.

The following figure shows the conversational relationship between a OpenEdge client and two DDE servers, Excel and a Visual Basic application. In this example, there are three open conversations—two with Excel and one with the Visual Basic application.

Figure 51: ABL DDE conversations



During a conversation, ABL can send and receive data between an ABL field or variable and a server data item, or send the value of an ABL expression to a server data item. Typically, you cannot access a server expression from ABL unless the expression is directly associated with a data item—for example, a worksheet expression that defines the value of a cell in Excel. In this case, while ABL can receive the value of the cell expression from the data item (cell), it cannot send a new value to that cell, since its value is determined by the server expression. However, DDE server applications, such as Excel, usually allow clients to completely redefine server topics and data items by executing server commands using the DDE. (The `i-ddeex1.p` procedure at the end of these topics demonstrates this capability.)

Defining conversation endpoints: DDE frames

ABL uses named frames as client endpoints for DDE conversations. A DDE frame, in effect, owns the client end of a conversation. You specify the DDE frame for a conversation when you open it with the `DDE INITIATE` statement.

A DDE frame can be visible or hidden during application execution. A hidden DDE frame effectively insulates the user from the conversation, preventing any unintended user intervention. However, the frame must be realized before you can open a conversation with it. That is, you must make it visible before hiding it (set the frame `VISIBLE` attribute = `TRUE`; then the `HIDDEN` attribute = `TRUE`).

A DDE frame can be a client endpoint for more than one DDE conversation, and you can use multiple DDE frames to open multiple conversations with the same application and topic. Your decision to use one or more DDE frames depends entirely on how you decide to allocate frames for your application and distribute DDE conversations among them. Otherwise, it makes no difference whether you use one or many DDE frames for conversation endpoints.

Each DDE frame provides a set of frame attributes that maintain the status of conversations owned by the frame. The following table lists these DDE frame attributes.

Table 57: DDE frame attributes

Attribute	Type	Readable	Setable
DDE-ERROR	INTEGER	✓	—
DDE-ID	INTEGER	✓	—
DDE-ITEM	CHARACTER	✓	—
DDE-NAME	CHARACTER	✓	—
DDE-TOPIC	CHARACTER	✓	—

Each time a conversational exchange occurs—that is, each time a DDE statement executes or ABL posts a DDE-NOTIFY event—ABL updates the DDE attributes of the frame that owns the conversation. These attributes are especially useful for determining the exact nature of a DDE run-time error (DDE-ERROR) and identifying the data item that triggered a DDE-NOTIFY event (DDE-NAME, DDE-TOPIC, and DDE-ITEM). (For more information on DDE-NOTIFY events, see Exchanging data in conversations on page 515.)

DDE-ERROR

This attribute records the error condition returned by the last exchange for the frame. The following table lists the possible errors returned by DDE conversation exchanges.

Table 58: ABL DDE errors

Error	Description
1	DDE INITIATE failure
2	A DDE statement (DDE ADVISE, EXECUTE, GET, REQUEST, or SEND) time-out
3	Memory allocation error
4	Invalid channel number (not an open conversation)
5	Invalid data item (in topic)
6	DDE ADVISE failure (data link not accepted)
7	DDE EXECUTE failure (commands not accepted)
8	DDE GET failure (data not available)
9	DDE SEND failure (data not accepted)

Error	Description
10	`DDE REQUEST` failure (data not available)
11	Data for `DDE-NOTIFY` event not available
99	Internal error (unknown)

Other DDE attributes

There are some other DDE attributes that you might need:

- **DDE-ID** — Records the channel number of the conversation that had the most recent exchange
- **DDE-ITEM** — Records the name of the server data item affected by the most recent conversation exchange
- **DDE-NAME** — Records the application name of the conversation that had the most recent conversation exchange
- **DDE-TOPIC** — Records the topic name of the conversation that had the most recent conversation exchange

Opening DDE conversations

This section discusses opening DDE conversations.

To open a DDE conversation in an ABL client:

1. Execute and prepare the server application to accept DDE conversations. This includes making all necessary server topics available to the client. This can be done either from inside the client or externally in Windows.
2. Define one or more named frames to use as DDE frames.
3. Open each conversation using the `DDE INITIATE` statement.

Preparing the server application

Before opening a DDE conversation from the client, you must ensure that the server application is open on the Windows desktop, and that any server preparations required to make application topics available to the OpenEdge client are complete. The OpenEdge client can open the server application by either by invoking the ABL `OS-COMMAND NO-WAIT` statement or by executing the `WinExec()` program load function from the Window's kernel dynamic link library (DLL), `kernel32.dll`. The program load function provides additional features that include specifying the startup window state and returning a value that indicates whether the application actually started.

For example, the code fragment in `i-ddeex2.p` defines the interface to `WinExec()`, and uses it to load Microsoft Notepad.

i-ddeex2.p

```
DEFINE VARIABLE ReturnCode AS INTEGER NO-UNDO.

PROCEDURE WinExec EXTERNAL "KERNEL32.DLL":
  DEFINE INPUT  PARAMETER ProgramName AS CHARACTER.
  DEFINE INPUT  PARAMETER VisualStyle AS LONG.
  DEFINE RETURN PARAMETER StatusCode  AS LONG.
END PROCEDURE.

/************************************************/
/* NOTE: VisualStyle parameters are as follows:   */
/*       1 = Normal         2 = Minimized         */
/************************************************/

RUN WinExec (INPUT "NOTEPAD", INPUT 1, OUTPUT ReturnCode).
IF ReturnCode >= 32 THEN
  MESSAGE "Application was Started" VIEW-AS ALERT-BOX.
ELSE
  MESSAGE "Application Failed:" ReturnCode VIEW-AS ALERT-BOX.

/*****************************************************************/
/* RETURN CODE DESCRIPTION                                       */
/* If the function is successful, the return value from WinExec  */
/* identifies the instance of the loaded module. Otherwise, the  */
/* return value is an error value between 0 and 32.              */
```

```
/*                                                               */
/*  0   System was out of memory, executable file was corrupt, or */
/*        relocations were invalid.                              */
/*  2    File was not found.                                     */
/*  3    Path was not found.                                     */
/*  5   Attempt was made to dynamically link to a task, or there */
/*         was a sharing or network protection error.           */
/*  6   Library required separate data segments for each task.   */
/*  8   There was insufficient memory to start the application.  */
/* 10    Windows version was incorrect.                          */
/* 11   Executable file was invalid. Either it was not a Windows */
/*        application or there was an error in the .EXE image.   */
/* 12   Application was designed for a different operating system. */
/* 13    Application was designed for MS-DOS 4.0.               */
/* 14    Type of executable file was unknown.                   */
/* 15    Attempt was made to load a real-mode application       */
/*         (developed for an earlier version of Windows).       */
/* 16   Attempt was made to load a second instance of an executable */
/*       file containing multiple data segments that were not   */
/*        marked read-only.                                     */
/* 19   Attempt was made to load a compressed executable file.  */
/*        The file must be decompressed before it can be loaded. */
/* 20   Dynamic link library (DLL) file was invalid.  One of the DLLs */
/*        required to run this application was corrupt.         */
/* 21   Application requies Microsoft Windows 32-Bit extensions. */
/*****************************************************************/
```

Depending on the application, the client might then open an initial conversation for the System topic and execute server commands to initialize additional topics.

Defining DDE frames

To open a conversation, you must first define a named frame to use as the DDE frame for the conversation. Make sure that you define the DDE frame in a scope large enough to complete the intended conversation. If you are unsure, use a FORM or DEFINE FRAME statement to define the frame for the scope of the procedure(s) that invoke DDE exchanges. For example, the following code defines and enables a frame for the procedure scope to start and manage conversations with Microsoft Excel:

```
DEFINE VARIABLE listx AS CHARACTER NO-UNDO
  VIEW-AS SELECTION-LIST SIZE 36 BY 5.
DEFINE VARIABLE ed    AS CHARACTER NO-UNDO
  VIEW-AS EDITOR SIZE 20 by 2.

DEFINE BUTTON bq LABEL "Quit".
DEFINE BUTTON bg LABEL "Start Excel".
...

DEFINE FRAME MainFrame
  SKIP(1) SPACE(1) bq SPACE(1) bg SPACE(1) SKIP(1)
  SPACE(1) listx LABEL "DDE History" SPACE(1) SKIP(1)
  SPACE(1) ed LABEL "Cell R4C2 (Row 4 Col B)" SKIP(1)
  WITH SIDE-LABELS.
ENABLE ALL WITH FRAME MainFrame TITLE "Main Frame".
...
```

If you want your application to completely hide DDE conversations from the user, always define your DDE frames as procedure-scoped static frames without fields and set their HIDDEN attributes to TRUE after realizing the frames. This prevents the user from doing anything that might compromise DDE communications, such as invoking an option that inadvertently destroys the frame in mid-conversation.

Initiating a conversation

After defining the DDE frame, invoke the DDE INITIATE statement for the frame, server name, and an available server topic. Most server applications provide a System topic that is available at startup. The data items for this topic generally include application command strings and other information to support DDE interactions with the application. In ABL, you can use this System topic to make other topics available to the client using the DDE EXECUTE statement. After you have made these topics available, you can open additional conversations for them.

The DDE INITIATE statement retrieves a channel number that uniquely identifies the conversation. You use this channel number with other DDE statements to invoke all other exchanges in the conversation.

For example, the following code fragment opens a conversation with Microsoft Excel's System topic, returning the channel number to sys. It uses the sys conversation to invoke the Excel NEW command, creating an initial Excel worksheet (Sheet1). It then opens a conversation with the Sheet1 topic, returning the channel number to sheet, as shown:

```
DEFINE VARIABLE sys   AS INTEGER NO-UNDO.
DEFINE VARIABLE sheet AS INTEGER NO-UNDO.
...
DDE INITIATE sys FRAME FRAME MainFrame:HANDLE
  APPLICATION "Excel" TOPIC "System".
DDE EXECUTE  sys COMMAND "[new(1)]".
DDE INITIATE sheet FRAME FRAME MainFrame:HANDLE
  APPLICATION "Excel" TOPIC "Sheet1".
...
```

Exchanging data in conversations

Once you have opened a DDE conversation, you can converse on the topic using four types of exchanges. These exchange types include:

- **Execute** — The DDE client executes server commands using the `DDE EXECUTE` statement.
- **Request** — The DDE client retrieves data from the server using the `DDE REQUEST` statement.
- **Send** — The DDE client sends data to the server using the `DDE SEND` statement.
- **Advise** — The DDE client creates *advise links* to server data items using the `DDE ADVISE` statement. Advise links direct the DDE server to monitor the data item. When any linked data item changes value, your OpenEdge application is notified so it can retrieve the value using the `DDE GET` statement.

Execute, request, and send exchanges are each *demand-driven*. When you invoke the exchange, ABL waits for the exchange to complete. A demand-driven exchange works like a `FIND` statement, which waits until ABL retrieves the desired record from the database. Advise exchanges are *event-driven*. Once an advise link is created for a data item, the OpenEdge client continues execution and retrieves the data item value "automatically" as it changes. You can invoke both demand, and event-driven exchanges in the same conversation and for the same data item. The following sections describe how to implement these exchanges.

Demand-driven exchanges

In a demand-driven exchange, the OpenEdge client sends data to or requests data from the server, and waits for each exchange to complete before continuing. The `DDE EXECUTE`, `DDE REQUEST`, and `DDE SEND` statements all invoke demand-driven exchanges.

Execute exchanges

You can use the `DDE EXECUTE` statement to send commands for the DDE server to execute, as shown in Initiating a conversation on page 514. Typically, you send server commands in a conversation opened for the System topic of the server application. However, the server might support commands for any topic that it provides. For example, Excel lets you select a range of cells (data items) in a worksheet topic. In effect, server commands are data items that it executes when sent with the `DDE EXECUTE` statement. For a more complete example of this type of exchange, see Closing DDE conversations on page 518.

The System topic for your server application might also provide other data items that you can read or set using request and send exchanges. These data items typically provide server options or status information to the client.

Request and send exchanges

You can use the `DDE REQUEST` statement to retrieve any server data item as a character string value. If you plan to convert the string value to another ABL data type (for example, `DATE`), you must convert the string to the correct format.

You can use the `DDE SEND` statement to send a character string value to any server data item. You must ensure that the string conforms to a format acceptable to the server data item.

You might use request and send exchanges to create new database records from the rows of a worksheet, or iteratively read a OpenEdge database and fill out worksheet rows from selected records. For example, the following code fragment fills out an Excel worksheet using three fields from the `Customer` table of the `sports2000` database:

```
DEFINE VARIABLE itemn AS CHARACTER NO-UNDO.
DEFINE VARIABLE rowi  AS INTEGER   NO-UNDO.
DEFINE VARIABLE sheet AS INTEGER   NO-UNDO.
...

DDE SEND sheet SOURCE "Name" ITEM "r1c1".
DDE SEND sheet SOURCE "Balance" ITEM "r1c2".
DDE SEND sheet SOURCE "State" ITEM "r1c3".

rowi = 2.
FOR EACH Customer NO-LOCK
  WHERE Customer.Balance > 10000 BY Customer.Balance:
  itemn = "R" + STRING(rowi) + "C1".
  DDE SEND sheet SOURCE Customer.Name ITEM itemn.
  itemn = "R" + STRING(rowi) + "C2".
  DDE SEND sheet SOURCE STRING(Customer.Balance) ITEM itemn.
  itemn = "R" + STRING(rowi) + "C3".
  DDE SEND sheet SOURCE STRING(Customer.State) ITEM itemn.
  rowi = rowi + 1.
END.
```

In the example, the first three DDE statements send column titles to the first three columns of the first row of the worksheet. Then, for each Customer record showing more than $10,000 in payables, the DDE statements send the customer's name, balance, and state of residence to the appropriate columns of the next row in the worksheet.

Event-driven exchanges

In an event-driven exchange, the DDE server application sends the value of the data item to the client whenever the value of the data item changes. Each server data item you set up for event-driven exchange actually participates in a series of exchanges, totally dependent on how often the data item changes. If the data item never changes value, no exchange occurs.

Steps to setting up event-driven exchanges - advise links

This section describes how to set up event-driven exchanges.

To set up and manage event-driven exchanges for a data item:

1. Specify the server data item in a `DDE ADVISE` statement, using the `START` option. This creates an advise link to the data item from ABL. From this point in the procedure, the DDE server monitors the specified data item, notifying ABL whenever its value changes.
2. Add an `ON DDE-NOTIFY OF FRAME`*`DDEframe`* statement, where *`DDEframe`* is the name of the frame that owns the conversation. In the trigger block for the statement, invoke the `DDE GET` statement to retrieve the new value of the data item (just like a `DDE REQUEST`) and process it as you want. ABL triggers this `DDE-NOTIFY` event and posts it to the appropriate DDE frame when notified of the value change. The OpenEdge client application then executes the event trigger when it blocks for I/O or invokes the `PROCESS EVENTS` statement. If you have more than one advise link established for the conversation, you can determine what data item changed by checking the value of the `DDE-ITEM` attribute of the DDE frame.

Note: In general, do not block for I/O (for example, invoke an UPDATE statement) or invoke a PROCESS EVENTS statement within the trigger for a DDE-NOTIFY event. This can cause ABL to update the DDE frame attributes for a new DDE-NOTIFY event before you have completed the processing for a prior event.

3. At any point in the procedure, if you want to stop event-driven exchanges for the data item, specify the data item in a DDE ADVISE statement using the STOP option. This removes the advise link and directs the server to cease monitoring value changes in the data item. This does not terminate the conversation in any way and you can continue to access the data item with other exchanges or create another advise link to the data item.

Coordinating DDE client/server communications

If you use event-driven exchanges, you might also want to set a value for the MULTITASKING-INTERVAL attribute of the SESSION system handle. The value of this attribute determines how often ABL checks for application events. An appropriate value can help your client application interact more smoothly with its DDE server. For more information, see *ABL Reference*.

Applications for event-driven exchanges

You might use event-driven exchanges to tie data item values in a OpenEdge application to those in a spreadsheet, calender scheduler, or some other database application that might be running in your application environment. For example, the following code fragment retrieves the latest value of the cell at row 4, column 2 (the "B" column) in an Excel worksheet and stores it in an ABL editor field:

```
DEFINE VARIABLE sheet AS INTEGER   NO-UNDO.
DEFINE VARIABLE sty   AS CHARACTER NO-UNDO.
DEFINE VARIABLE ed    AS CHARACTER NO-UNDO VIEW-AS EDITOR SIZE 20 by 2.

ON DDE-NOTIFY OF FRAME MainFrame DO:
  DDE GET sheet TARGET sty ITEM "r4c2".
  sty = SUBSTR(sty, 1, 20).           /* Drop the CR/LF */
  ed:VALUE IN FRAME MainFrame = sty.
END.
...
DDE ADVISE sheet START ITEM "r4c2".
...
```

Multiple conversations with advise links

If a DDE frame owns more than one conversation with advise links, you can get the channel number of the conversation and the name of the data item to which a DDE-NOTIFY event applies from the DDE-ID and DDE-ITEM attributes of the frame. This is enough information to retrieve the value that triggered the event with the DDE GET statement. However, you might also need the application and topic names to fully identify the item from which you are retrieving data. This is necessary if the different conversations (different topics and/or applications) include links to data items with the same name. You can obtain the application and topic names related to the current DDE-NOTIFY event from the DDE-NAME and DDE-TOPIC attributes of the frame. For more information on DDE frame attributes, see Defining conversation endpoints: DDE frames on page 510.

However, you might find it simpler to manage event-driven exchanges by using a separate DDE frame for each advise link. In this case, you initiate an additional conversation for each advise link that you want to establish for the same application and topic, using a separate DDE frame for each conversation. You then set up each advise link using its own conversation ID. When a `DDE-NOTIFY` event occurs for a linked data item, you do not have to check for the source of the event, because each data item has its own frame trigger. The following code fragment shows the essential elements of this technique to link two data cells in the same Excel worksheet to two ABL variables (`quote1` and `quote2`):

```
DEFINE VARIABLE link1  AS INTEGER   NO-UNDO.
DEFINE VARIABLE link2  AS INTEGER   NO-UNDO.
DEFINE VARIABLE quote1 AS CHARACTER NO-UNDO.
DEFINE VARIABLE quote2 AS CHARACTER NO-UNDO.

DEFINE FRAME Flink1.
DEFINE FRAME Flink2.

ON DDE-NOTIFY OF FRAME Flink1 DO:
  DDE GET link1 TARGET quote1 ITEM "R2C1".
  ...
END.

ON DDE-NOTIFY OF FRAME Flink2 DO:
  DDE GET link2 TARGET quote2 ITEM "R2C2".
  ...
END.

DDE INITIATE link1 FRAME Flink1:HANDLE APPLICATION "EXCEL" TOPIC "Sheet1".
DDE INITIATE link2 FRAME Flink2:HANDLE APPLICATION "EXCEL" TOPIC "Sheet1".

DDE ADVISE link1 START ITEM "R2C1".
DDE ADVISE link2 START ITEM "R2C2".
```

Closing DDE conversations

You can close a DDE conversation in one of three ways:

- Invoke the `DDE TERMINATE` statement for the conversation
- Leave the scope of the DDE frame that owns the conversation
- Terminate or remove the DDE server or server topic associated with the conversation

Regardless of how you close a DDE conversation, once closed, the channel number for that conversation is no longer available for further exchanges. Terminating the conversation with the `DDE TERMINATE` statement has no effect on other conversations open for the same DDE server or frame. They continue without interruption.

In general, if your OpenEdge client creates and manages a server environment for the conversation, it should also clean up that environment when closing the conversation. For example, the following code fragment cleans up a Microsoft Excel server environment when the user presses a **QUIT** button. The client removes an advise link to the worksheet and closes the conversation for that worksheet topic (sheet). Then using the System conversation (`sys`), it executes server commands that close the documents opened for the conversation, clear Excel error checking, and instruct the server to shut itself down. For example:

```
DEFINE VARIABLE sys   AS INTEGER NO-UNDO.
DEFINE VARIABLE sheet AS INTEGER NO-UNDO.

DEFINE BUTTON bq LABEL "Quit".
...

ON CHOOSE OF bq IN FRAME MainFrame DO:
  ...
  DDE ADVISE sheet STOP ITEM "r4c2".
  DDE TERMINATE sheet.
  DDE EXECUTE sys COMMAND "[activate(~"sheet1~")]".
  DDE EXECUTE sys COMMAND "[close(0)]" 0).
  DDE EXECUTE sys COMMAND "[activate(~"chart1~")]".
  DDE EXECUTE sys COMMAND "[close(0)]".
  DDE EXECUTE sys COMMAND "[error(0)]".
  DDE EXECUTE sys COMMAND "[quit()]".
END.
...
```

Note: The tilde (~) is used to escape embedded quotes in strings, for example, `~"sheet1~"`.

DDE example

The `i-ddeex1.p` ABL procedure uses the DDE facility to build a Microsoft Excel worksheet of `Customer` balances from the `sports2000` database and display the `Customer` payables distribution in an Excel pie chart. It incorporates most of the code examples in the previous sections.

The visible DDE frame displays a selection list (DDE History) showing the server, topic, and item name for each exchange as it occurs. (Note the custom selection list delimiter (|) used in place of the default comma delimiter that appears in commands.) When you manually change the balance value in cell `r4c2` of the worksheet, ABL uses an advise link to retrieve and display the new value in the field labeled `Cell R4C2 (Row 4 Col B)`.

Note: For `i-ddeex1.p` to work, you must have Excel in your path or specify the full Excel pathname for the `prog_name` parameter when you run the WinExec procedure to start Excel. This example assumes that you have Excel in the default Microsoft Office directory.

i-ddeex1.p

```
DEFINE VARIABLE listx   AS CHARACTER NO-UNDO
  VIEW-AS SELECTION-LIST SIZE 36 BY 5.
DEFINE VARIABLE rowi    AS INTEGER   NO-UNDO.
DEFINE VARIABLE sys     AS INTEGER   NO-UNDO.
DEFINE VARIABLE sheet   AS INTEGER   NO-UNDO.
DEFINE VARIABLE itemn   AS CHARACTER NO-UNDO.
DEFINE VARIABLE sty     AS CHARACTER NO-UNDO.
DEFINE VARIABLE ed      AS CHARACTER NO-UNDO VIEW-AS EDITOR SIZE 20 by 2.
DEFINE VARIABLE log_i   AS LOGICAL   NO-UNDO.
DEFINE VARIABLE excelon AS LOGICAL   NO-UNDO.

DEFINE BUTTON bq LABEL "Quit".
DEFINE BUTTON bg LABEL "Start Excel".

DEFINE FRAME MainFrame
  SKIP(1) SPACE(1) bq SPACE(1) bg SPACE(1) SKIP(1)
  SPACE(1) listx LABEL "DDE History" SPACE(1) SKIP(1)
  SPACE(1) ed LABEL "Cell R4C2 (Row 4 Col B)" SKIP(1)
  WITH SIDE-LABELS.

ENABLE ALL WITH FRAME MainFrame TITLE "Worksheet Monitor".
listx:DELIMITER = "|". /* No server commands use "|" */

PROCEDURE WinExec EXTERNAL "kernel32.dll": /* Run Windows application */
  DEFINE INPUT PARAMETER prog_name  AS CHARACTER.
  DEFINE INPUT PARAMETER prog_style AS LONG.
END PROCEDURE.

PROCEDURE LogExchange: /* Log latest DDE in selection list */
  log_i = listx:ADD-LAST(FRAME MainFrame:DDE-NAME + " " +
    FRAME MainFrame:DDE-TOPIC + " " +
    FRAME MainFrame:DDE-ITEM) IN FRAME MainFrame.
END PROCEDURE.

ON CHOOSE OF bq IN FRAME MainFrame DO:
  IF (excelon = FALSE) THEN RETURN.
  DDE ADVISE sheet STOP ITEM "r4c2". RUN LogExchange.
  DDE TERMINATE sheet. RUN LogExchange.
  DDE EXECUTE sys COMMAND "[activate(~"sheet1~")]". RUN LogExchange.
  DDE EXECUTE sys COMMAND "[close(0)]". RUN LogExchange.
  DDE EXECUTE sys COMMAND "[activate(~"chart1~")]". RUN LogExchange.
  DDE EXECUTE sys COMMAND "[close(0)]". RUN LogExchange.
  DDE EXECUTE sys COMMAND "[error(0)]". RUN LogExchange.
  DDE EXECUTE sys COMMAND "[quit()]". RUN LogExchange.
END.
```

```
ON CHOOSE OF bg IN FRAME MainFrame DO:
  /* INPUT: 1=normal 2=minimized */
  RUN WinExec ("C:\Program Files\Microsoft Office\Office\Excel /e", INPUT 2).
  excelon = TRUE.

  DDE INITIATE sys FRAME FRAME MainFrame:HANDLE
    APPLICATION "Excel" TOPIC "System". RUN LogExchange.
  IF sys = 0 THEN DO:
    MESSAGE "Excel not available".
    RETURN.
  END.
  DDE EXECUTE sys COMMAND "[new(1)]". RUN LogExchange.
  DDE INITIATE sheet FRAME FRAME MainFrame:HANDLE
    APPLICATION "Excel" TOPIC "Sheet1". RUN LogExchange.
  DDE SEND sheet SOURCE "Name" ITEM "r1c1". RUN LogExchange.
  DDE SEND sheet SOURCE "Balance" ITEM "r1c2". RUN LogExchange.
  DDE SEND sheet SOURCE "State" ITEM "r1c3". RUN LogExchange.
  rowi = 2.

  FOR EACH Customer NO-UNDO
    WHERE Customer.Balance < 5000 BY Customer.Balance:
    itemn = "R" + STRING(rowi) + "C1".
    DDE SEND sheet SOURCE Customer.Name ITEM itemn. RUN LogExchange.
    itemn = "R" + STRING(rowi) + "C2".
    DDE SEND sheet SOURCE STRING(Customer.Balance) ITEM itemn.
      RUN LogExchange.
    itemn = "R" + STRING(rowi) + "C3".
    DDE SEND sheet SOURCE STRING(Customer.State) ITEM itemn. RUN LogExchange.
    rowi = rowi + 1.
  END.

  DDE EXECUTE sheet COMMAND "[select(~"C1:C2~")]". RUN LogExchange.
  DDE EXECUTE sys COMMAND "[column.width(,,,3)]". RUN LogExchange.
  DDE EXECUTE sys COMMAND "[format.number(~"$#,##0~")]". RUN LogExchange.
  DDE EXECUTE sheet COMMAND "[select(~"C1:C2~")]". RUN LogExchange.
  DDE EXECUTE sys COMMAND "[new(2,1,1)]". RUN LogExchange.
  DDE EXECUTE sys COMMAND "[activate(~"chart1~")]". RUN LogExchange.
  DDE EXECUTE sys COMMAND "[gallery.3d.pie(3)]". RUN LogExchange.
  DDE EXECUTE sys COMMAND "[app.restore()]". RUN LogExchange.
  DDE EXECUTE sys COMMAND "[arrange.all()]". RUN LogExchange.

  DDE ADVISE sheet START ITEM "r4c2". RUN LogExchange.
END. /* ON CHOOSE of bg IN FRAME MainFrame */

ON DDE-NOTIFY OF FRAME MainFrame DO:
  DDE GET sheet TARGET sty ITEM "r4c2". RUN LogExchange.
  sty = SUBSTR(sty, 1, 20).             /* Drop the CR/LF */
  ed:SCREEN-VALUE IN FRAME MainFrame = sty.
END.

WAIT-FOR CHOOSE OF bq IN FRAME MainFrame.
```

16

Using COM Objects in ABL

Both ActiveX Automation objects and ActiveX controls are COM objects—objects that conform to the specifications of the Microsoft Component Object Model (COM). As such, COM objects have common features that govern the programming of both Automation objects and ActiveX controls in ABL (Advanced Business Language).

These topics describe the common features of COM objects supported by OpenEdge and how to work with them in an ABL application and your development environment.

For an overview of the architectural elements that support COM objects in OpenEdge, see Introduction to External Program Interfaces on page 413. For information on COM object support that is unique for Automation objects, see ActiveX Automation Support on page 543 and for ActiveX controls, see ActiveX Control Support on page 551. The information in these topics applies equally to Automation objects and ActiveX controls, except where noted.

For details, see the following topics:

- How COM objects differ from ABL widgets
- Obtaining access to COM objects
- Accessing COM object properties and methods
- Managing COM objects in an application
- Locating COM object information on your system

How COM objects differ from ABL widgets

COM objects and ABL widgets are two different types of objects. Although they can provide similar capabilities for an application, their function and management in ABL is quite different.

Functionality

COM objects are provided by third-party vendors as well as by Progress Software Corporation. They have standard features that allow them to be accessed from many different programming environments. The capabilities of COM objects vary widely. Many provide some user interface component, while others provide such functionality as HTTP support and internet access. However, because COM objects are independent of ABL, they have no direct integration with OpenEdge data management facilities, such as database fields, formats, and validation.

ABL widgets are objects that are completely defined by ABL and are not directly accessible by any other programming environment. Their capabilities focus primarily on user interface function. Unlike COM objects, ABL widgets are fully integrated into the programming and data management facilities of ABL. This means that a variety of default ABL behaviors apply to widgets that do not apply to COM objects.

ABL mechanisms

Ultimately, the differences between COM objects and widgets require separate but related mechanisms in ABL to work with them. The most fundamental of these mechanisms are the handles you use to access COM objects and widgets. You must access widgets with widget handles and COM objects with component handles. These handles have different data types that represent the different capabilities that they support—the `HANDLE` data type for widget handles and the `COMPONENT-HANDLE` (or `COM-HANDLE`) data type for COM objects.

`COM-HANDLE` values have a data type that is compatible with most other ABL data types. For component handles, ABL does the data conversion automatically, unlike widget handles which require the use of ABL data conversion functions, like `STRING` or `INTEGER`.

The following table compares each type of object (COM object and ABL widget) and summarizes how it is supported in ABL.

Table 59: COM objects and ABL widgets compared

Feature	COM object support	ABL widget support
Compilation	No compile-time checking	Compile-time checking for attribute and method references
Instantiation	Dynamic, with different mechanisms for Automation objects and ActiveX controls	Static or Dynamic, depending on the widget and application
Handle type	`COM-HANDLE`	`HANDLE`
Handle validity	`VALID-HANDLE` function	`VALID-HANDLE` function
Access to data	Properties (optionally indexed)	Attributes
Access to behavior	Methods (with unnamed parameters)	Methods (with unnamed parameters)
Access to events	Event procedures (with unnamed parameters)	Triggers
Integration with ABL data	Extensive automatic mapping between COM data types and ABL data types	Full integration with ABL data types and ABL data management

Feature	COM object support	ABL widget support
Color management	RGB color values (RGB-VALUE function, GET-RGB-VALUE method on the COLOR-TABLE system handle)	Index numbers of RGB color values in ABL color table (COLOR-TABLE system handle)
Dynamic object management	RELEASE OBJECT statement	DELETE Widget statement

The remainder of the topics in this section describe the equivalent mechanisms that support COM objects.

Obtaining access to COM objects

To access a COM object, you have to define a component handle for it and set the handle value using the appropriate object instantiation. For example:

```
DEFINE VARIABLE hCOMobject AS COM-HANDLE.
/* ... Instantiate COM object, setting hCOMobject ... */
/* ... Access COM object properties and methods using hCOMobject ... */
/* ... Access ActiveX control events using OCX event procedures ... */
```

To instantiate an Automation object, you use the CREATE Automation Object statement. For more information, on instantiating Automation objects, see ActiveX Automation Support on page 543.

To instantiate an ActiveX control, you use the AppBuilder at design time to select and configure the control, and to generate ABL that instantiates the control at runtime. This AppBuilder-generated ABL includes the CREATE Widget statement to create the control-frame widget and the LoadControls() method to associate the control instance with the control-frame COM object. For more information on instantiating ActiveX controls, see ActiveX Control Support on page 551.

Access to COM object properties and methods is the same for both Automation objects and ActiveX controls. ABL supports event management for both Automation objects and ActiveX controls. For information on ActiveX Automation object event management, see ActiveX Automation Support on page 543. For information on ActiveX control event management, see ActiveX Control Support on page 551.

Accessing COM object properties and methods

ABL supports direct access to COM object properties and methods. This support extends the syntax used for widget attribute and method references. Other than syntax validation, ABL does no compile-time checking of property and method references. All other property and method validation occurs at runtime, when all COM objects are (dynamically) instantiated. Thus at runtime, ABL dispatches each property or method reference to the COM object for evaluation and execution.

Property and method syntax

The syntax to access properties and methods is similar to the syntax to access widget attributes and methods. However, where widget references use widget handles, COM object references use component handles. Component handles support an extended syntax that allows you to:

- Chain component handle references to properties and methods

- Specify indexes on properties
- Specify mappings between ABL data types and COM data types for method output parameters and property settings
- Specify additional options for method parameters and return values

The syntax diagrams that follow describe the syntax for method and property references. These diagrams are equivalent to the syntax presented in the *ABL Reference*, but describe method and property references in a more top-down fashion. (See the information on attributes and methods in *ABL Reference*.)

Note: All COM object errors are translated to ABL errors. To suppress any error messages generated by a COM object method or property reference, you can specify the NO-ERROR option in any statement that includes the method or property reference.

Method reference

This is the syntax for a COM object method reference:

Syntax

```
[ NO-RETURN-VALUE ]
COMhdl-expression:Method-Name-Reference
```

NO-RETURN-VALUE

Required for some methods that do not have a return value.

NO-RETURN-VALUE prevents ABL from expecting a possible return value. This option is appropriate only if the method does not have a return value. Whether a method call requires this option depends on the COM object. Some COM objects require that the caller knows there is no return value. In this case, you must specify the option. Many more robust COM objects do not require this option.

If the method requires NO-RETURN-VALUE and you don't specify it, the COM object generally returns an error. For example, Word for Windows 95 Version 7.0 returns an error similar to "Non-function called as function."

Note: If the method has a return value, you must not invoke it in a statement with the NO-RETURN-VALUE option. In this case, ignore the return value without specifying this option, as in the second call to the SetDate() method.

COMhdl-expression

An expression that returns a component handle to the COM object that owns the method specified by *Method-Name-Reference*.

Method-Name-Reference

Specifies a single COM object method that might return a value.

Property reference

This is the syntax for a COM object property reference:

Syntax

```
COMhdl-expression:Property-Name-Reference
```

COMhdl-expression

An expression that returns a component handle to the COM object that owns the property specified by *Property-Name-Reference*.

Property-Name-Reference

Specifies a single COM object property.

Component handle expression

Every method or property reference must begin with a component handle expression that returns a component handle value. This is the syntax to specify a component handle expression:

Syntax

```
COMhandle [ :Method-Name-Reference | Property-Name-Reference ] . . .
```

COMhandle

A component handle variable. (Note that the first element in a component handle expression must be a `COM-HANDLE` variable.)

Method-Name-Reference

Specifies a single COM object method or property that returns a component handle value. A component handle expression can chain as many method and property references as required to return the handle to a particular COM object.

Property-Name-Reference

Specifies a single COM object method or property that returns a component handle value. A component handle expression can chain as many method and property references as required to return the handle to a particular COM object.

Note: For the most efficient dispatch of multiple references to a particular COM object, assign the initial component handle expression for the COM object to a `COM-HANDLE` variable. Each reference to a given component handle expression incurs additional run-time overhead for each method or property referenced.

Method name reference

This is the syntax to specify a single COM object method:

Syntax

```
Method-Name
  (
    { [ OUTPUT | INPUT-OUTPUT ]
          expression [ AS datatype ]
          [ BY-POINTER | BY-VARIANT-POINTER ]
      | null-parameter
    }
    [ , [ OUTPUT | INPUT-OUTPUT ]
          expression [ AS datatype ]
          [ BY-POINTER | BY-VARIANT-POINTER ]
      | null-parameter
    ]...
  )
```

Method-name

The name of the COM object method. This name is not case sensitive. But by convention, OpenEdge documentation shows COM object method names in mixed case.

expression

Any valid ABL expression that you can pass as a parameter to the method.

datatype

One of several data-type specifiers that the associated parameter might require. For more information on *datatype*, see Table 60: Data-types for COM object methods and properties on page 530. The remaining keyword options specify additional type and mode information for each parameter. The COM object defines the data types and numbers of parameters for a method.

null-parameter

Any amount of white space, indicating an optional parameter that you choose to omit. You can also pass variable numbers of parameters if the method supports it. For more information on specifying method parameter options, see Specifying options for properties and method parameters on page 531.

You can invoke a method in two ways:

- Include the appropriate method reference as part of an ABL expression. This expression can be on the right side of an assignment statement or in any other statement that accepts the expression. Methods invoked

with this technique must return a value. For example, if `YearDay()` is a method that returns the number of days from the first of the year to a specified date, it can appear in the following bolded expressions:

```
DEFINE VARIABLE myObject    AS COM-HANDLE NO-UNDO.
DEFINE VARIABLE day-of-year AS INTEGER    NO-UNDO.

/* ... Access COM object as myObject ... */

day-of-year = myObject:YearDay("05/01/1997").

DISPLAY myObject:YearDay(TODAY) " days have passed this year.".
```

- Specify the appropriate method reference as a statement, ignoring any return value. Methods invoked as a statement might require the `NO-RETURN-VALUE` option (shown previously in the Method Reference syntax). If `SetDate( )` allows you to set a date for a COM object, you might invoke it in the following bolded statements:

```
DEFINE VARIABLE myObject    AS COM-HANDLE NO-UNDO.
DEFINE VARIABLE day-of-year AS INTEGER    NO-UNDO.

/* ... Access COM object as myObject ... */

NO-RETURN-VALUE myObject:SetDate(1995,5,1). /* Set "May 1, 1995" */
myObject:SetDate(,5,1). /* Set "May 1" of current year */
```

Property name reference

This is the syntax to specify a single COM object property:

Syntax

```
Property-Name [ ( index [ , index ] ... ) ]
   [ AS datatype-specifier ]
```

Property-Name

The name of the COM object property. This name is not case sensitive. But by convention, OpenEdge documentation shows COM object property names in mixed case.

index

Any expression that legally indexes the property and for the required number of dimensions. If necessary and if possible, ABL converts the data type of *index* to the COM data type expected by the property. Essentially, the syntax is the same as for a method reference with input parameters.

datatype-specifier

One of several data-type specifiers that the associated property might require when you set its value. For more information on *datatype*, see Table 60: Data-types for COM object methods and properties on page 530. For more information on specifying the AS *datatype* option and data type conversion for properties, see Specifying options for properties and method parameters on page 531.

You can reference a property in two ways:

- Read the property by specifying the appropriate property reference as part of an ABL expression. This expression can be on the right side of an assignment statement or in any other statement that accepts the expression, as in the following bolded expressions:

```
DEFINE VARIABLE myObject AS COM-HANDLE NO-UNDO.
DEFINE VARIABLE degF     AS INTEGER    NO-UNDO.

/* ... Access COM object as myObject ... */

degF = myObject:TempFahrenheit.
DISPLAY "The current temperature is " myObject:TempFahrenheit " degF.".
```

- Write (set) the property by specifying the appropriate property reference on the left side of an assignment statement, as in the following bolded property references:

```
DEFINE VARIABLE myObject AS COM-HANDLE NO-UNDO.

/* ... Access initial COM object as myObject ... */

myObject:SavingsAcount("1000-24-369"):AccountName(1) = "John".
myObject:SavingsAcount("1000-24-369"):AccountName(2) = "Doe".
```

  Note the use of a chained component handle expression to reference a savings account object. `AccountName` is an indexed property of the `SavingsAccount` object that specifies a first and last name for the account.

Data-type specifier

ABL supports a protocol to provide default mappings between native COM object method parameter and property values and the corresponding ABL data values. This mapping protocol supports many COM data types. You can override the default mapping by using a type specifier, as shown in the following table. For more information on mapping data types shown in the following table, see COM Object Data Type Mapping on page 609.

Table 60: Data-types for COM object methods and properties

Data-type specifier to override the default mapping	ABL data type
CURRENCY	DECIMAL
ERROR-CODE	INTEGER
IUNKNOWN	COM-HANDLE
FLOAT	DECIMAL
SHORT	INTEGER
UNSIGNED-BYTE	INTEGER

Each of these data-type specifiers represents one of the base COM data types available for a method parameter or property. Each ABL data type represents the typical ABL data type that corresponds to the specified COM data type.

Data-type specifier options are necessary only for some method input parameters and property settings. Whether or not you must specify the AS *datatype* or any of the other type options (`BY-POINTER` or `BY-VARIANT-POINTER`) depends on the COM object method or property and the implementation of the COM object. You can also specify how a method parameter of any data type is passed using the mode options `OUTPUT` or `INPUT-OUTPUT`. Whether you use a mode option depends (in part) on how you plan to use the method parameter in your application. For more information on using data-type specifiers and mode options, see Specifying options for properties and method parameters on page 531, and COM Object Data Type Mapping on page 609.

Specifying options for properties and method parameters

ABL allows you to specify a variety of data-type specifier and mode options for passing COM object method parameters and setting COM object properties (see Property and method syntax on page 525). Data-type specifier options specify a data type mapping between COM data types and ABL data types; mode options specify how a method parameter is passed (whether for input or output). Thus:

- The data-type specifier options in Table 60: Data-types for COM object methods and properties on page 530 dictate COM data type conversions for passing method input parameters and setting properties that are different than the defaults. (For more information on COM data-type conversion, see COM Object Data Type Mapping on page 609)
- The type options `BY-POINTER` and `BY-VARIANT-POINTER` specify additional information for passing method parameters.
- The mode options `OUTPUT` and `INPUT-OUTPUT` specify how the method parameter is used.

One of the essential criteria that determines when and how you might have to use data-type specifier and mode options is the Type Library provided with a COM object implementation.

Understanding a COM object Type Library

A *Type Library* contains definitions for a COM object's methods and properties. When a COM object provides a Type Library, ABL references it before dispatching the method or property in an attempt to convert each method parameter or property value to the required COM data type. If a Type Library is available, ABL tries to match the number and types of any parameters being passed into a method before dispatching the method to the COM object for execution.

If both data-type specifier options and Type Library definitions are provided, the data-type specifier options take precedence. For more information on how OpenEdge matches ABL data items to COM object properties and method parameters, see COM Object Data Type Mapping on page 609.

You can locate data type information that is stored in Type Libraries on-line using the OpenEdge COM Object Viewer. For more information on Type Libraries and how to view their components, see Locating COM object information on your system on page 538.

Using data-type specifier options

Data-type specifier options allow you to be more specific about how to convert information from the OpenEdge application into the data type expected by the COM object. These data-type specifiers override ABL default data type conversions for COM object properties and method parameters that have no Type Library support. For information on ABL default data conversions and how they are affected by data-type specifiers and Type Library support, see COM Object Data Type Mapping on page 609.

Using data-type specifiers for properties

You can include a data-type specifier option in a property reference on the left side of an assignment statement, when you set the property. You can use any data-type specifier from Table 60: Data-types for COM object methods and properties on page 530. For example:

```
myObject:Money AS CURRENCY = 1000.9999.
```

Using data-type specifiers for method parameters

For a method parameter, you can also use a data-type specifier from Table 60: Data-types for COM object methods and properties on page 530. For COM objects that do not have Type Library definitions and yet require that the data type of the parameter be passed properly, you must specify the data type for the method call to succeed. For example:

```
myObject:SaveMoney(1000.9999 AS CURRENCY).
```

In addition, you can use the BY-POINTER or BY-VARIANT-POINTER type option to indicate that the parameter is to be passed as a pointer. For example:

```
myObject:SaveMoneyPtr(1000.9999 AS CURRENCY BY-POINTER).
```

A *pointer* is a value that contains the memory location of the data item you are referencing. BY-POINTER specifies a pointer to the data item value. BY-VARIANT-POINTER specifies a Variant that contains a pointer to another Variant that stores the actual value.

Note: A Variant is a self-describing COM data type. It contains both the data and an indication of its effective data type.

Both the BY-POINTER and BY-VARIANT-POINTER options have no affect on the value of the parameter. They only affect how the data is packaged when the method is dispatched to the COM object. The type option to use, if any, is determined by the method implementation. You can determine the type options required for each parameter from the OpenEdge COM Object Viewer (see Locating COM object information on your system on page 538).

Using mode options for method parameters

The default mode for a method parameter is input. An input parameter passes a value to the method but does not return a value from the method. Thus, an input parameter can be a database field, an expression, or a variable.

Caution: Do not use the INPUT keyword as a mode option because, for a parameter, ABL might interpret this as the screen value of a widget.

The mode option OUTPUT or INPUT-OUTPUT specifies a parameter that returns a value from the method. (An INPUT-OUTPUT parameter also passes a value to the method.) This means that the value of any passed variable can change after the method call returns. You can only specify the OUTPUT or INPUT-OUTPUT options with a variable as the parameter (as opposed to a database field or an expression):

```
DEFINE VARIABLE MyWallet AS DECIMAL NO-UNDO.

myObject:WithdrawMoneyPtr(OUTPUT MyWallet AS CURRENCY).
```

Note: The OUTPUT or INPUT-OUTPUT option forces the parameter to be passed as a pointer and explicitly specifies that a value be returned to your application. Thus, if you use a mode option, you do not need to use the BY-POINTER type option because the type option is redundant. However, the BY-POINTER type option, by itself, does not return a value to your program. You must use a mode option or ABL does not allow the method to return a value in the parameter.

Note that ABL does not use Type Library information to determine the parameter mode. This prevents the COM object from updating a variable that you do not expect or want to change. Thus, if the COM object ordinarily changes the value of a particular parameter, you can prevent any variable you pass from having its value changed by omitting any mode option on the parameter.

Restrictions on property and method references

ABL supports most features necessary to reference COM object properties and methods. However, there are some restrictions:

- Parentheses are required for method calls, whether or not they take parameters.
- There is no support for named (keyword) parameters.
- There is no support for default properties or methods.

Parentheses on method calls

In general, you must specify all method references with parentheses. Although some COM objects accept a method call referenced without parentheses, ABL is not aware that such a reference is a method call. Without parentheses, the compiler interprets the reference as a property rather than as a method. This can cause unpredictable results for the method call.

Named method parameters

The COM standard allows named method parameters that you can specify in any order for a method call. ABL does not support named parameters in any form. The first line of this example shows an illegal named parameter, where the Filename parameter is passed to the SaveAs method on an Excel Workbook handle:

```
Excel-Workbook-hdl:SaveAs(Filename = "x.xls"). /* Illegal syntax */
Excel-Workbook-hdl:SaveAs("x.xls").            /* Legal syntax */
```

The second line shows the form that ABL accepts. In general, whether or not you omit optional parameters, you must pass all method parameters in the correct parenthesized order.

Default properties and methods

The COM standard allows for default properties and methods. For any COM object, the default property or method is invoked when you reference only the COM object handle. For example, **Item** is the default (indexed) property for collection objects. However, ABL requires that you specify all properties and methods you want to reference, whether or not they are defaults for the COM object. For example:

```
DEFINE VARIABLE hCollection AS COM-HANDLE NO-UNDO.

/* ... Set hCollection to a collection COM object ... */
hCollection:Item(1). /* This is legal in ABL. */
hCollection(1).      /* This is not legal in ABL. */
```

Managing COM objects in an application

ABL provides a number of mechanisms in ABL to help manage COM objects in an application. Some of these mechanisms derive from general ABL constructs previously available, while others are added to ABL just for COM object support. Together, they support the following application tasks:

- Component handle validation
- Font and color management
- ActiveX collection navigation
- Resource management
- Error handling

Validating component handles

If you reference a `COM-HANDLE` variable whose value does not point to a valid COM object, ABL returns an error indicating that an action was performed on an invalid `COM-HANDLE`. To protect against this error, use the ABL `VALID-HANDLE` function to determine if the `COM-HANDLE` variable contains a valid value before using it in any other ABL statement. As with widget handles, the `VALID-HANDLE` function returns `TRUE` if the component handle is valid.

Note that you cannot use the `VALID-HANDLE` function to verify that a component handle value points to a particular COM object. `COM-HANDLE` values might be reused within an application when the COM objects they point to are no longer available. For more information, see Managing COM object resources on page 536.

Also, this function only indicates that a component handle is invalid from some action (or inaction) of the OpenEdge application. It does not show as invalid a COM handle that a user might have manually closed, for example, an Automation Server application that provided the COM object.

Managing COM object font and color settings

You can manage both fonts and colors for COM objects from ABL. However, ABL provides more direct support for color than for font management.

Managing fonts

Most Automation Servers or ActiveX controls that support font manipulation provide an associated Font object to manage font changes. As such, you can use Font object properties and methods to read or set font values for the COM object. For information on font support, see the documentation on your Automation Server or ActiveX control. Otherwise, there is no mapping between COM object font settings and the ABL font information maintained by the FONT-TABLE system handle.

Managing colors

COM objects accept color specifications in the form of an RGB (Red/Green/Blue) integer value. However, ABL widgets accept color specifications in the form of an integer index into a color table managed by the COLOR-TABLE system handle. To support color management for COM objects, ABL provides techniques that work with or without the ABL color table.

ABL provides three ways to obtain a color value to set colors for a COM object:

- Use the value from the property or method of another COM object.
- Use the GET-RGB-VALUE( ) method of the COLOR-TABLE system handle.
- Use the RGB-VALUE function.

To use a color value from another COM object, simply assign the color value returned by one of its properties or methods to a color property or as a method parameter of your Automation object or ActiveX control.

To use the GET-RGB-VALUE( ) method, pass it an index to a color stored in the ABL color table and the method returns an integer that represents the RGB value of the specified color. You can then assign this value to a COM object color property. For example:

```
myObject:BorderColor = COLOR-TABLE:GET-RGB-VALUE(5).
```

This example assigns the RGB value of color number 5 from the color table to the BorderColor property of myObject.

To use the RGB-VALUE function, you pass the function three color values between 0 and 255 and it returns a single RGB value that represents the color. For example:

```
myObject:BorderColor = RGB-VALUE(127, 255, 75). /* Red, Green, Blue */
```

Navigating ActiveX collections

ActiveX collections are COM objects that reference multiple instances of a particular class of COM object. ABL supports collection navigation by allowing you to access a COM object through the indexed **Item** property of the collection object. For example:

```
DEFINE VARIABLE ix       AS INTEGER    NO-UNDO.
DEFINE VARIABLE ExcelApp AS COM-HANDLE NO-UNDO.

/* Instantiate Automation object for Excel.Application in ExcelApp */

CREATE "Excel.Application" ExcelApp.
. . .
REPEAT ix = 1 TO ExcelApp:Sheets:Count():
  ExcelApp:Sheets:Item(ix):Name = "ABC" + STRING(ix).
END.
```

In this example, ABL loops through all the worksheet objects in the Excel Application collection. Sheets is the collection and **Item** is the indexed property that returns the component handle to each Sheet object. The code uses the index (i) to loop through the total number of Sheet objects in the collection (`ExcelApp:Sheets:Count()`), and assigns a unique name to each one ("ABC1", "ABC2", and so on).

Note: Collections are often named as a plural of the object class that they index. Thus, in this example, Sheets is the collection class and Sheet is the object class whose instances are indexed by the **Item** property of Sheets.

Managing COM object resources

When working with COM objects, especially Automation objects, it can be very easy to instantiate many object instances (for example, when searching many objects in a collection). This accumulation of COM objects can impose a burden on system resources. To alleviate this burden, ABL provides the `RELEASE OBJECT` statement. This statement releases the COM object associated with the specified component handle, making it available for garbage collection if no other COM object has a reference to it. In general, it is good practice to release any COM handles that you no longer need for your application. It is only necessary to release COM objects that have been assigned to a component-handle variable.

Releases and deletes

Note that you release COM objects, but delete widgets. A release is different from a delete because, by convention, a COM object stays around until there are no other references to it. In the case of an ActiveX control, the control-frame is a COM object that references the control (see Introduction to External Program Interfaces on page 413). If you set a component handle variable to the control, this is a second reference. However, no matter how many component handle variables you set to the same control, this represents a single reference from ABL. (ABL takes care of this for you.) If you release the control through a component handle while its control-frame is still instantiated, the control remains instantiated because the control-frame COM object still references the control. In this case, you must delete the control-frame widget to finally free the ActiveX control. In a similar way, multiple COM object references can also keep an Automation object and its resources tied up longer than necessary.

Release strategy

In general, to maximize the reusability of COM object resources, always release a COM object when you no longer need it. Because of references between COM objects, the status of references to a particular COM object might not always be obvious (especially for Automation objects). By releasing COM objects as soon as you know they are not needed in an application, you have the best chance of ensuring that their resources are available for reuse as soon as possible.

Releases and component handles

When you release a COM object, this invalidates every component handle that references the same object. Any further attempt to use an invalidated component handle results in an error indicating that an invalid action was performed on an invalid COM-HANDLE value.

On the other hand, if you instantiate a different COM object after releasing the first one, a previously invalidated component handle might point to the new COM object. The component handle might be reused because the object server might reuse memory left over from the released COM object. In this case, the component handle might be seen as valid. However, the object that the component handle references is different, so this might result in errors or successful method calls with results that are invalid for your application.

In general, to avoid errors and confusion with invalidated COM handles, it is a good practice to set any COM-HANDLE variables to the Unknown value (?) once you have released it.

Note: ABL maintains no relationship between one component handle and another, such as when you derive one component handle from a property value on another component handle. The two component handles are completely independent of each other. This is different from the parent/child relationships that some COM objects (especially Automation objects) might maintain among themselves. In this case, invoking the right method on a "parent" COM object might well trigger the release of many other "related" COM objects. For information on any such cascading object relationships, see the documentation on a particular Automation Server or ActiveX control.

Error handling

ABL traps all COM object errors caused by property and method references. ABL formats the error information into an ABL error message that includes the hexadecimal code of the COM object error and explanatory text. If you specify the NO-ERROR option on a statement that references a COM object property or method, ABL stores the error information in the ERROR-STATUS handle, as shown:

```
DEFINE VARIABLE chMyObject AS COM-OBJECT NO-UNDO.
DEFINE VARIABLE ix         AS INTEGER    NO-UNDO.

/* ... Instantiate COM Object with chMyObject ... */
chMyObject:Method(10) NO-ERROR.
IF ERROR-STATUS:NUM-MESSAGES > 0 THEN
DO ix = 1 TO ERROR-STATUS:NUM-MESSAGES:
  MESSAGE ERROR-STATUS:GET-MESSAGE(ix).
END.
```

Note: Some types of ABL statements treat any errors as warnings even if the method or property reference results in a serious error. For warnings, ABL does not set the ERROR-STATUS:ERROR attribute. Thus, to detect that any exception (warning or error) occurred, you must check ERROR-STATUS:NUM-MESSAGES for a value greater than zero, as in the example.

You cannot otherwise access COM object error information directly unless a method includes this information in its return value or in an output parameter. This means you cannot reliably respond to a particular COM object error code. You can only tell that an error occurred and search the message text for a set of likely error codes.

Locating COM object information on your system

You can view the COM objects provided with any Automation Server or ActiveX control that is installed on your system by using the OpenEdge COM Object Viewer. This tool parses the Type Library installed with a selected Automation Server or ActiveX control to provide three lists:

- Automation objects that are *createable* (usable in the CREATE Automation Object statement) or ActiveX controls that are selectable as OCXs in the AppBuilder
- All COM objects supported by the selected Automation Server or ActiveX control
- The properties, methods, and events for any selected COM object

Using the COM Object Viewer

To use the COM Object Viewer:

1. Choose **Tools** > **PRO*Tools** in any ADE window.
2. Click the **COM Object Viewer** icon (third icon from the right) in the **PRO*Tools** toolbar:



   To execute the Viewer from Windows Explorer, navigate to your `%DLC%\bin` installation directory and double-click the `proobjvw.exe` icon.

   To execute the Viewer from the command line, open an MS-DOS Prompt window and enter `proobjvw.exe` at the prompt.

3. Open the Type Library for an Automation Server or ActiveX control.
4. Locate objects in the Viewer.
5. Review, cut, and paste in your application any available syntax to use an object.

Accessing Type Libraries

Type Libraries describe most COM objects, including both Automation objects and ActiveX controls. A Type Library can exist in one of the following forms:

- A separate file with the extension `.tlb` or `.olb`. This file is usually in the same directory as the main binary file for the Automation Server or ActiveX control.
- Part of the main binary file for the Automation Server or ActiveX control, generally with the extension `.exe`, `.dll`, or `.ocx`. This is often the form provided with ActiveX controls.

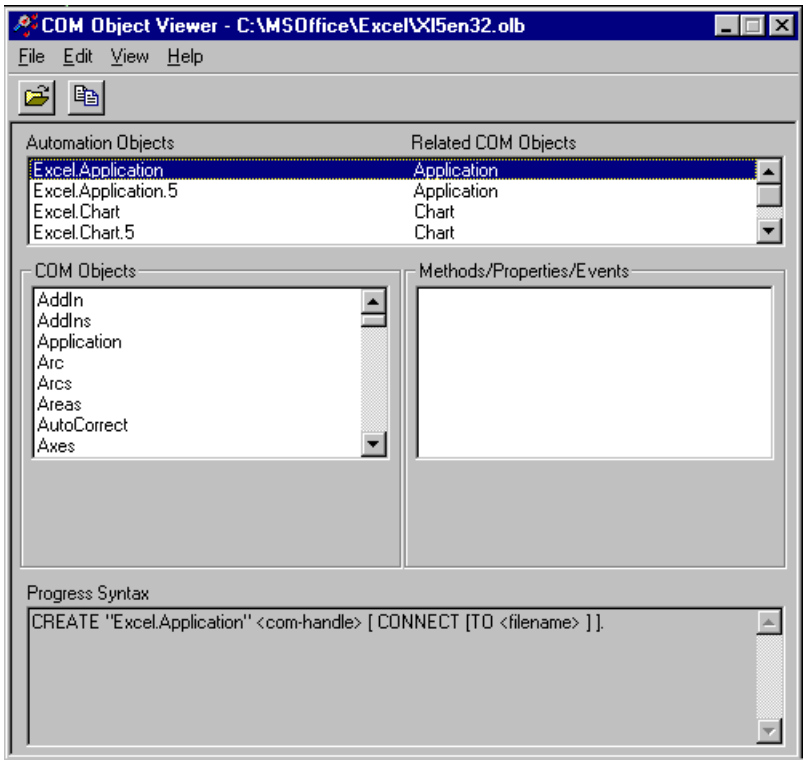
Locating objects in the Viewer

The top listbox label in the Viewer changes depending on whether you open an Automation Server Type Library or an ActiveX control Type Library. For Automation Servers, the top listbox is Automation Objects. For ActiveX controls it is Controls.

Automation objects

For Automation Servers, the Viewer displays all createable Automation objects in the Automation Objects listbox, as shown in the following figure.

Figure 52: Automation objects in the COM Object Viewer



In COM, a createable Automation object has an identifier known as a Program Identifier (ProgID in the registry). This identifier is the expression that you use to identify the Automation object in the ABL CREATE Automation Object statement (see ActiveX Automation Support on page 543). The Automation Objects listbox lists the ProgID of each createable Automation object followed by the corresponding COM object (Related COM Objects).

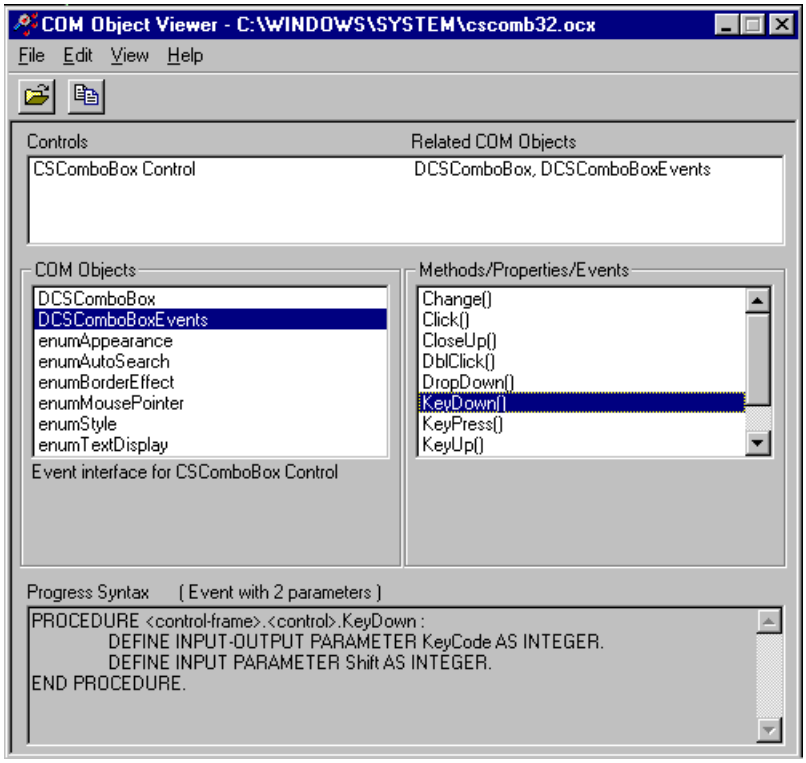
When you select an item in the Automation Objects listbox, an OpenEdge Syntax editbox at the bottom of the window shows sample ABL syntax for creating it. You can cut and paste this syntax into an ABL procedure.

The Viewer lists all COM objects that are available from the Automation Server to an Automation Controller, like OpenEdge, in the COM Objects listbox. In general, only a small number of Automation objects are createable. You then use the properties and methods on these COM objects to access the other COM objects listed for the Server. You cannot determine the relationship among COM objects from the OpenEdge COM Object Viewer tool. For more information on this, see the documentation provided with each Automation Server.

ActiveX controls

For ActiveX controls, the Viewer displays the name of the control in the Controls listbox that is selectable as an OCX in the AppBuilder, as shown in the following figure.

Figure 53: ActiveX controls in the COM Object Viewer



The listed control name is the OCX name (not the control-frame name) that the AppBuilder displays for the control when you select and insert it in a design window. This name is followed by the names of corresponding COM objects (Related COM Objects).

When you select an item in the Controls listbox (not shown selected), no syntax appears in the OpenEdge Syntax editbox at the bottom of the window. This is because the AppBuilder generates all required syntax for creating an ActiveX control in your application at runtime.

The viewer lists all COM objects that are available to ABL with this control in the COM Objects listbox.

In general, only one listed object is available in the AppBuilder at design time, the ActiveX control, itself. You then use the properties, methods, and events on the ActiveX control to access the other COM objects listed in the COM Objects listbox at runtime. (For more information on ActiveX control events, see ActiveX Control Support on page 551) You cannot determine the relationship among COM objects from the OpenEdge COM Object Viewer tool. For more information on this, see the documentation provided with each ActiveX control.

Viewing methods, properties, and events

When you select a COM object in the COM Objects listbox, a Methods/Properties/Events listbox shows the methods, properties, and events of the COM object in alphabetical order. Methods and events appear with a pair of parentheses following the name. Properties for constant COM objects appear as a set of constant values.

When you select an item in the Methods/Properties/Events listbox, the OpenEdge Syntax editbox shows sample ABL syntax for using it. You can cut and paste this syntax into an ABL procedure. Figure 53: ActiveX controls in the COM Object Viewer on page 540 shows the OCX event procedure syntax for the CSComboBox KeyDown event.

The tool also displays information on the method, property, or event above the OpenEdge Syntax editbox, including:

- The number of parameters for the method, indexed property, or event
- Whether the property is read-only
- Whether a constant value is being displayed

With the sample ABL syntax, the tool displays any important information on parameters and return data type, including:

- Methods that do not return a value (called with `NO-RETURN-VALUE`)
- The return data type of a method that does return a value (shown prefixed to the name of the sample variable on the left side of the assignment)
- The data type of the parameters (shown prefixed to the name of each sample variable for a parameter)
- The mode of the parameter (`OUTPUT` or `INPUT-OUTPUT`)
- The value of a constant
- Optional portions of the syntax in brackets ([...])
- `PROCEDURE` definitions for events, including any parameters

17

ActiveX Automation Support

ActiveX Automation allows one application (the *Automation Controller*) to drive another application (the *Automation Serve*r) through COM objects (*Automation objects*) provided by the Automation Server. An Automation object thus represents both a package of Server functionality and a point of connection from the Automation Controller to the Automation Server. This functionality can include many capabilities from exchange of data between the two applications to almost total control of the Server application by the Controller.

OpenEdge supports ActiveX Automation as an Automation Controller. This allows you to write an ABL (Advanced Business Language) application that connects to and drives Automation objects in an Automation Server. You do this by creating a connection to the Automation object and referencing its properties and methods.

These topics describe how to build an ABL application that functions as an Automation Controller, featuring a working example. It relies on ABL support for COM object access in ABL. For information on ABL support for COM objects, see Using COM Objects in ABL on page 523.

For details, see the following topics:

- Requirements for doing Automation
- Accessing Automation Servers
- Managing Automation objects
- Automation event support

Requirements for doing Automation

The main requirement for implementing Automation in an application is that the Automation Server is correctly installed on the system where you develop and deploy the application.

To write an Automation application:

1. Access an Automation Server by creating connections to one or more createable Automation objects provided by the Server (using the CREATE Automation Object statement).
2. Access data and functionality on the Server and instantiate additional Automation objects as needed, through the properties, methods, and events of the created Automation objects.
3. Enable events for the Automation object, if desired.
4. Release each Automation object as your application no longer needs them.
5. Repeat Steps 1 through 3 for as many Automation Servers as your application requires.

The possible combinations of Servers and Automation objects used by your application are limited only by the available resources and implementation of the Automation Servers.

Note that Automation objects can be organized into hierarchies in an Automation Server, with one or more top-level Automation objects providing access to the rest. Top-level objects represent the application for the Server and are always createable. Other Automation objects in the Server can also be createable. The relationship of each Automation object in the hierarchy affects how you can instantiate it. For more information on the relationship among Automation objects, see the documentation on their Automation Server.

Accessing Automation Servers

The CREATE Automation Object statement provides four basic connection options to access an Automation Server. (For more information, see the entry for the CREATE Automation Object Statement in *ABL Reference*.) Each option handles different connection requirements, depending on the Automation object implementation and the execution status of the Automation Server. Any Automation object flagged in the registry with REGCLS_MULTIPLEUSE launches only a single instance of the Server that handles all subsequent object instantiation requests. Any Automation object flagged in the registry with REGCLS_SINGLEUSE launches a Server dedicated to that object instance. Any subsequent Automation object instantiated for the same Server launches a new instance of the Server dedicated to that object.

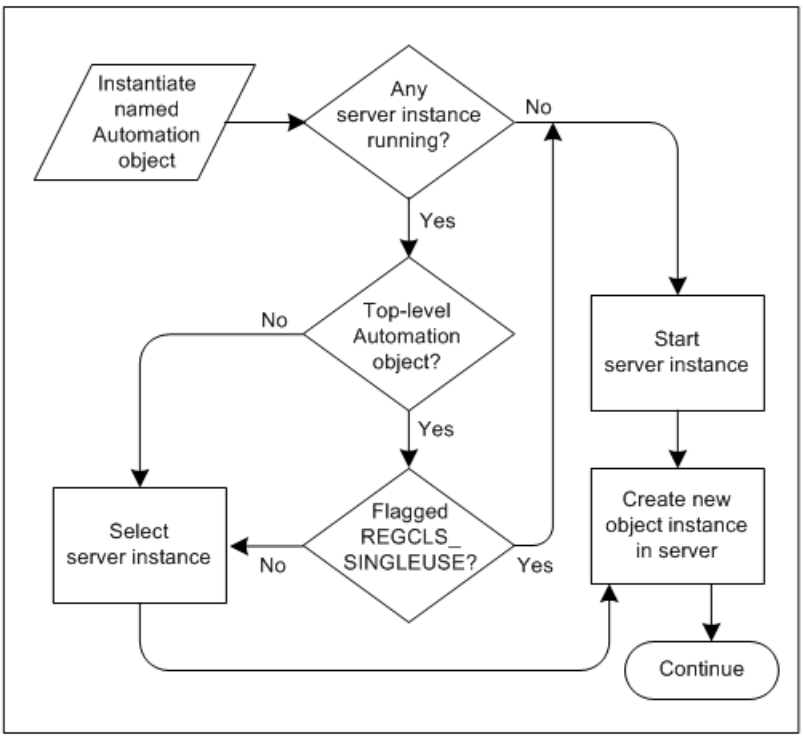
Option 1: Instantiate Automation object by name

This option creates a connection to a new instance of a specified Automation object, launching the Server if necessary. For top-level Automation objects (such as Excel.Application), this option usually launches a new instance of the Server, unlike for lower-level objects (such as Excel.Sheet, which use the instance created by a top-level object). For example:

```
DEFINE VARIABLE hExcelObject AS COM-HANDLE NO-UNDO.

CREATE "Excel.Application" hExcelObject.
```

The following figure summarizes the basic logic for this option.

Figure 54: Automation connection option 1



Note: This option is equivalent to the `Visual Basic CreateObject(`*`class`*`)` or `GetObject("",`*`class`*`)` function call.

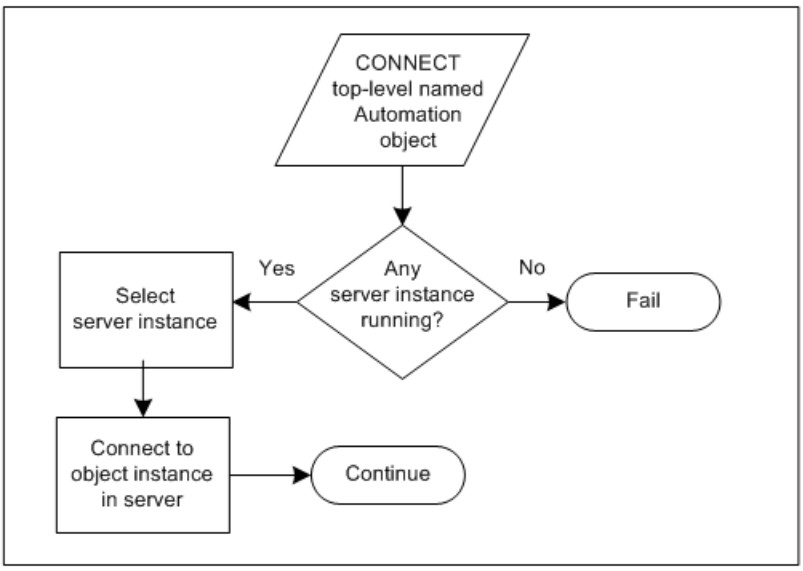
Option 2: Connect to top-level named Automation object

This option creates a connection to an existing instance of a top-level Automation object (such as "Excel.Application"). This does not work with lower-level objects (such as "Excel.Sheet"), and fails if the Server is not already running. For example:

```
DEFINE VARIABLE hExcelObject AS COM-HANDLE NO-UNDO.

CREATE "Excel.Application" hExcelObject CONNECT.
```

The following figure summarizes the basic logic for this option.

Figure 55: Automation connection option 2



Note: This option is equivalent to the Visual Basic `GetObject(,`*`class`*`)` function call.

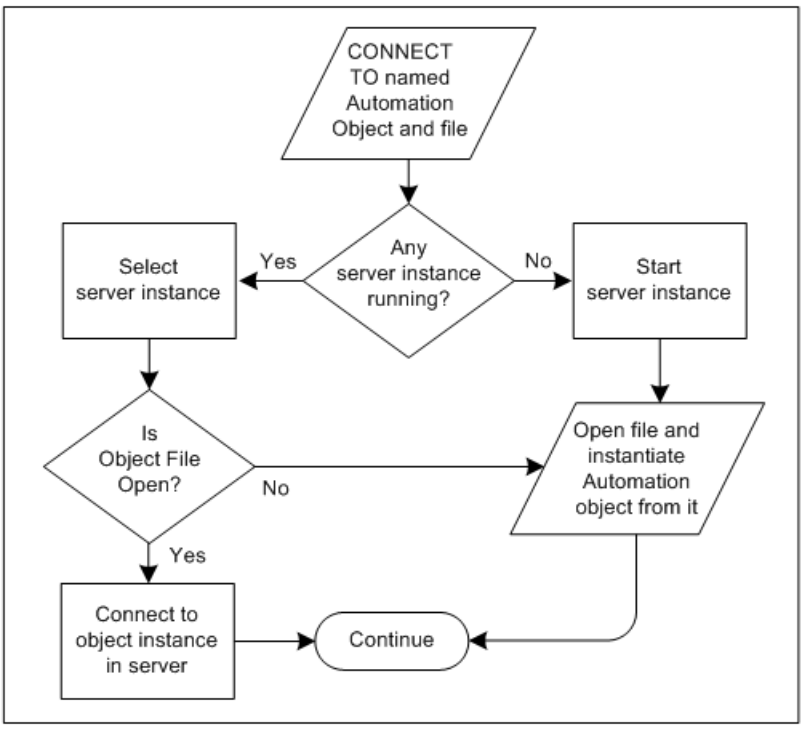
Option 3: Connect to or instantiate a named Automation object and file

This option creates a connection to a new or existing instance of the specified Automation object and opens the specified file. If the file is not already open, it is opened. If the pathname for the specified file is invalid or unrecognizable by the Server, this connection option fails. In this example, the file `\WorkSheets\Xplan.xls` is opened in the new or existing instance of the "Excel.Sheet" object:

```
DEFINE VARIABLE hExcelObject AS COM-HANDLE NO-UNDO.

CREATE "Excel.Sheet" hExcelObject CONNECT TO "\WorkSheets\Xplan.xls".
```

The following figure summarizes the basic logic for this option (ignoring the listed error conditions).

Figure 56: Automation connection option 3



Note: This option is equivalent to the Visual Basic `GetObject(`*`pathname`*`,`*`class`*`)` function call.

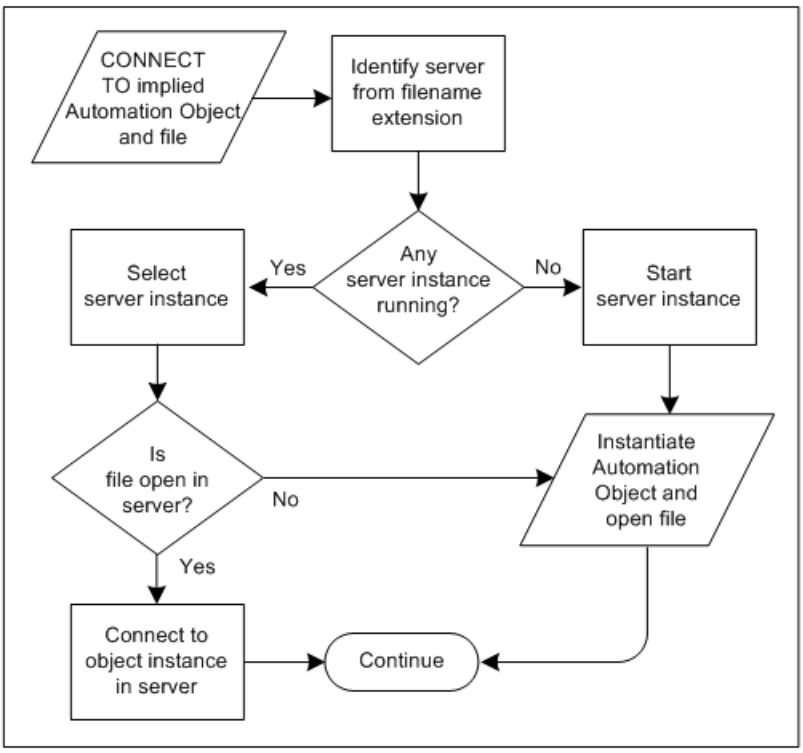
Option 4: Connect to or instantiate implied Automation object and file

This option creates a connection to a new or existing instance of the Automation object implicitly defined by the specified file. This option identifies the Automation object and its Server from the specified filename extension, as defined in the registry. If the file is not already open in the Automation Object, it will be opened. In this example, the `.xls` extension indicates that the object instance is a Sheet object provided by the Excel Automation Server:

```
DEFINE VARIABLE hExcelObject AS COM-HANDLE NO-UNDO.

CREATE "" hExcelObject CONNECT TO "\WorkSheets\Xplan.xls".
```

The following figure summarizes the basic logic for this option (ignoring the listed error conditions).

Figure 57: Automation connection option 4



Note: This option is equivalent to the Visual Basic `GetObject(`*`pathname`*`)` function call.

Managing Automation objects

ABL provides the `RELEASE OBJECT` statement to release Automation objects that your application no longer requires. Efficient use of resources requires that you actively manage the Automation objects you instantiate in your application. Each object remains active until there are no remaining references from the Server or other Automation Controllers and one of the following events occurs:

- The OpenEdge session terminates
- You actively release the Automation object

To efficiently manage Automation, you must release all Automation objects that you instantiate, either directly (through the `CREATE` Automation Object statement) or indirectly (through properties and methods of other Automation objects). There is no association between Automation objects that automatically propagates the release of one from the release of another.

In general, always release an Automation object when you are certain no other functionality in your application requires it. Note that if more than one component handle variable references the same Automation object, releasing the object with one handle invalidates them all.

Caution: To avoid misleading errors while developing your application, set all equivalent component handles to the Unknown value (?) after releasing Automation objects on any one of them.

For more information on managing Automation objects, see the sections on COM object management in Using COM Objects in ABL on page 523.

Automation event support

ABL supports event notification for ActiveX Automation objects with a built-in method that enables event notification for that specific component handle.

Note: This method is invalid when the component handle represents a control, since events work for controls automatically.

The following syntax describes the ENABLE-EVENTS method for ActiveX Automation objects:

Syntax

```
ENABLE-EVENTS ( event-proc-prefix )
```

event-proc-prefix

A CHARACTER expression representing a string prepended to every event received. The resulting string is the name of the internal procedure ABL runs in response to an event notification. During an event notification, all running procedures and all persistent procedures are searched to find a procedure with the name matching *event-proc-prefix.eventname* (for example, ExcelWB.SelectionChanged).

For more information on managing Automation objects, see the sections on COM object management in Using COM Objects in ABL on page 523.

For information on ActiveX control event management, see ActiveX Control Support on page 551.

18

ActiveX Control Support

An *ActiveX control (OCX)* is a reusable component built on the Microsoft Component Object Model (COM) that allows you to extend the user interface and functionality of a OpenEdge application. Some controls are similar to ABL (Advanced Business Language) widgets, such as combo boxes and radio sets. There are a wide variety of additional user interface controls that are not available as built-in ABL widgets, such as spin buttons, various dialogs, meters, and picture controls. There are also nongraphical controls for such tasks as communications and time keeping that have no user interface.

These topics describe the mechanics of working with ActiveX controls in ABL. It does not fully explain how to use the OpenEdge AppBuilder to incorporate ActiveX controls in an application. The information in these topics relies on ABL support for COM objects. For information on COM objects in ABL, see Using COM Objects in ABL on page 523.

For details, see the following topics:

- How ABL supports ActiveX controls
- Creating a control instance in ABL
- Orienting a control in the user interface at design time
- Accessing ActiveX controls at runtime
- Managing ActiveX controls at runtime
- Handling events
- Programming ActiveX controls in the AppBuilder
- ActiveX controls and examples installed with ABL

How ABL supports ActiveX controls

In general, ActiveX control technology is a more powerful, flexible, and robust replacement for the VBX control standard. To comply with the COM standard, an ActiveX control instance must be placed in a *control container* that handles events and specific user-interface functionality for the control. ABL supports this standard with the control-frame widget, a field-level widget and an associated control-frame COM object (which provides the actual control container support). For an overview of ActiveX controls, how they interact with ABL, and the requirements for using them in ABL, see Introduction to External Program Interfaces on page 413

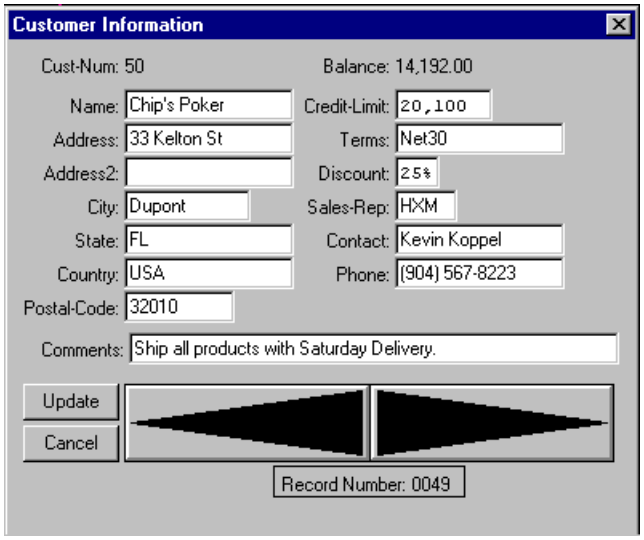
You can extend a OpenEdge application with numerous commercially available ActiveX controls. In ABL, you must use the AppBuilder at design time to create instances of ActiveX controls in your application. You can then write ABL code to reference ActiveX control properties and methods and to define event handlers that respond to ActiveX control events.

An example ActiveX control in ABL

The following figure shows a spin button control (the large left- and right-arrow buttons) implemented in the example application, `i-ocx1.w`. This spin button control overlays a portion of the frame space defined by an ABL control-frame widget.

Note: This spin button control is one of three ActiveX controls provided with your OpenEdge installation. The other two are combo box and timer controls. For more information on these controls, see ActiveX controls and examples installed with ABL on page 577.

Figure 58: ActiveX control example



In this example, the control-frame (and thus, the control itself) has been largely stretched in the horizontal direction and slightly stretched in the vertical to present a convenient access point in the dialog box. A spin button control is generally designed to provide incremental and decremental values, like a slider. This application uses the events generated by spin button value changes to scan back and forth through a list of `Customer` records in the `sports2000` database. It uses the incremental and decremental control values to maintain a record count (Record Number field). Handling events on page 565 explains how ABL applications handle events and values.

How ABL encapsulates an ActiveX control

ABL encapsulates an ActiveX control using the control-frame. The control-frame provides the basic interface between the ActiveX control and ABL through its two separate but related objects, the control-frame widget and the control-frame COM object. (For an overview of the relationship between ActiveX controls and the control-frame, see Introduction to External Program Interfaces on page 413)

Control-frame widget

The control-frame widget is a field-level widget, and as such it establishes the relationship between the ActiveX control and other ABL widgets in the user interface. Thus, it is the control-frame widget attributes (like `ROW` and `TAB-POSITION`) that maintain the relationship between the ActiveX control and other field-level widgets. For example, the spin button control in Figure 58: ActiveX control example on page 552 has a location and tab order that is determined by the control-frame widget location and tab order in the **CustomerInformation** dialog box.

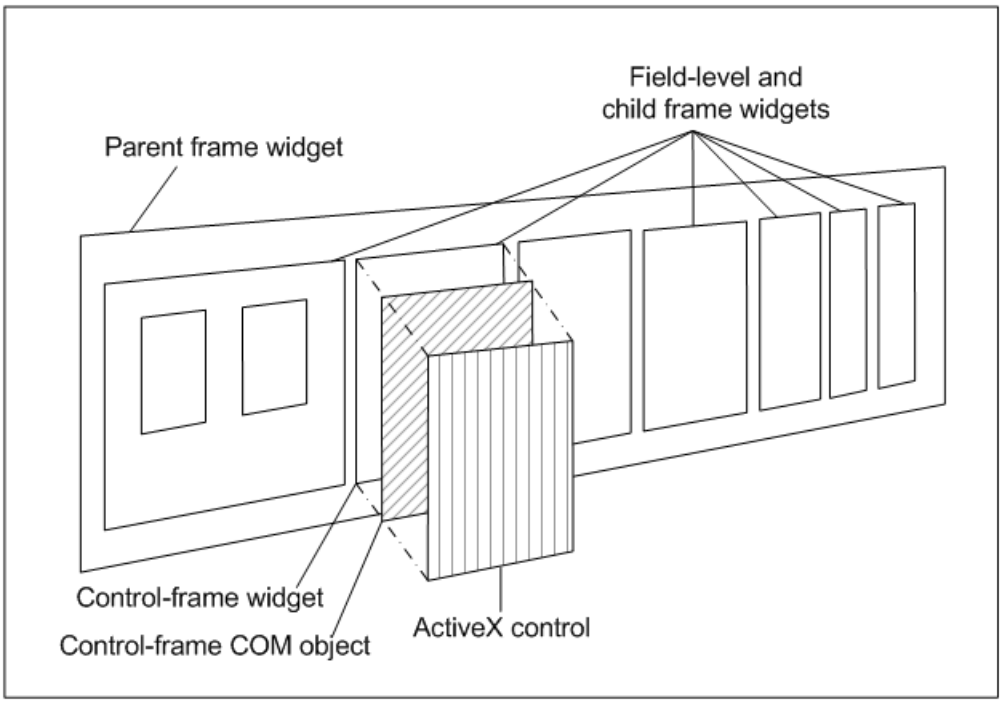
The control-frame widget also allows you to handle widget events that interact with other field-level widgets (like `TAB` and `LEAVE`) or that have special ABL significance (like `GO` and `END-ERROR`).

Control-frame COM object

The control-frame COM object is the actual ActiveX control container. It provides the initial point of access to the ActiveX control from ABL. Through this COM object you can access the component handle of the ActiveX control, which in turn allows you to directly access the ActiveX control's properties and methods. For information on accessing ActiveX control properties and methods in ABL, see Using COM Objects in ABL on page 523.

The following figure shows the relationship between the ActiveX control, its control-frame, and other widgets in the ABL user interface.

Figure 59: ActiveX control encapsulation in ABL



Control-frame attributes and properties

In addition to the control-frame widget attributes that manage an ActiveX control's relationship to the user interface, the control-frame COM object has several properties that map to corresponding control-frame widget attributes. These properties provide another way of getting to the same information and are listed in the following table.

Table 61: Control-frame attributes and properties

Widget attribute	COM object property
`HEIGHT-PIXELS, HEIGHT[-CHARS]`	Height
`NAME`	Name
`WIDTH-PIXELS, WIDTH[-CHARS]`	Width
`X, COLUMN`	Left
`Y, ROW`	Top

If one of these control-frame widget attributes changes, the corresponding COM object property changes, and the reverse is also true. For example, suppose `CtrlFrame` is the widget handle to a control-frame and `chCtrlFrame` is the component handle to the same control-frame. If you set `CtrlFrame:X = 10`, the value of chCtrlFrame:Left is set to 10. If you set `chCtrlFrame:Width = 100` (100 pixels), the value of `CtrlFrame:WIDTH-PIXELS` is set to 100. In this case, `CtrlFrame:WIDTH-CHARS` also changes, but the exact value (number of characters) depends on the font.

For a complete list of the attributes and properties (as well as methods and events) associated with the control-frame, see the `CONTROL-FRAME` Widget entry in *ABL Reference*.

Additional control functionality

An ABL control container often maintains information regarding the management of the controls it contains. This information is available as an extended control which essentially provides a wrapper around the base control. Even though the extended control maintains the extended properties, methods, and events, each of these appears as though it is part of the base control. As a result, accessing any extended feature is identical to accessing a feature on the base control itself. An example of an extended property is the property Tag, as its functionality is provided by the container. The base control does not even know the property Tag exists. It is the extended control that handles any access to the Tag property.

Properties

The following table lists the available extended properties.

Table 62: Extended properties

Property name	Type	Access
HonorProKeys	logical	read/write at design timeread/write at runtime
Name	string	read/write at design timeread/write at runtime

Property name	Type	Access
Parent	COM-handle	read at runtime
Tag	string	read/write at design timeread/write at runtime
Visible	logical	read/write at design timeread/write at runtime

The following table defines each property and how it can be used.

Table 63: Property definitions

Property	Definition
HonorProKeys	The default value of TRUE allows the ABL code to process the **GO**, **ENDKEY**, **HELP**, or **TAB** key stroke as each is defined. Setting the value to FALSE causes the control to process the key stroke without ABL code receiving notification that the key stroke occurred.
Name	The Name property contains the name of the control. The name is important because it identifies the control. You can use the control's name to get a `COM-HANDLE` to the control (for example, `chCSSpin=chCFSpin:CSSpin`, where `CSSpin` is the control's name and `chCFSpin` is the control frame handle). The control name associates event handlers with a control.
Parent	The Parent property is the com-handle (a pointer to the IDispatch interface) of the container in which the control resides. This property is set internally by ABL.
Tag	The Tag property is a user property that allows the user to store an arbitrary string value and retrieve it later. ABL does not use this property internally, and it is intended to give the user a way of storing application specific information with the control. This property is initialized to an empty string.
Visible	The Visible property determines and indicates whether an ActiveX control is currently displayed. The Visible property is distinct from, but influenced by, the Visible and Hidden attributes of the Control-Frame widget. The Visible property will appear in the Property Editor and can be set at design time. It defaults to TRUE. The value set in the Property Editor determines whether the OCX is initially displayed when the program is run, but can be overriden by the value of the Control-Frame widget's Hidden attribute.Some ActiveX controls are never displayed at runtime (for example, a timer control.) For these controls, the Visible property will not appear in the Property Editor and attempts to set the property at runtime will have no effect.

Special considerations for extended controls

The control container normally maintains relationships among controls. However, in ABL, because the control-frame can hold only a single ActiveX control, the extended properties (such as those that manage control position within the control container) can be maintained on the control-frame widget itself. Thus, the control-frame widget attributes manage the relationship between the ActiveX control and the parent frame widget (its effective container) in the user interface. For this same reason, additional extended properties that would otherwise be available on the extended control are found on the control frame widget instead.

ActiveX control restrictions

A single control-frame is capable of holding only a single ActiveX control. However, you can include many control-frames (and thus many ActiveX controls) in an ABL frame. This limitation incurs certain restrictions on the types of ActiveX controls that you can use. For instance, there is no internal support for container controls—that is, controls that allow multiple controls to reside inside of them.

Creating a control instance in ABL

You must use the AppBuilder to add an ActiveX control to your application.

To add a control in the AppBuilder:

1. Open a container object, such as a Dialog or SmartWindow.
2. Select an ActiveX control from the object Palette by clicking the **OCX** button to open the **Choose Control** dialog box, or by clicking the button for one of the three ABL-supported ActiveX controls.
3. Drop the selected control into your design window.

Note: These are general steps.

If you save your work at this point, the AppBuilder generates the following data and code for your application:

- The definition for a default instance of the specified ActiveX control in a binary (`.wrx`) file. This includes the initial definition provided by the control vendor. The binary file includes all ActiveX control instances for the corresponding `.w` file.
- Default ABL code in your `.w` file to instantiate and orient the ActiveX control in the user interface at runtime.

The rest of this section describes how you can modify the default definition of an ActiveX control instance. For more information on the ABL code for instantiating an ActiveX control at runtime, see Accessing ActiveX controls at runtime on page 559. For more information on locating and sizing the control in the user interface, see Orienting a control in the user interface at design time on page 558.

Understanding design time and runtime properties

To customize the definition for an ActiveX control, you must change the values of control properties in the AppBuilder at design time. Each ActiveX control supports a specific list of properties that you can set at design time (*design time properties*) and another list of properties that you can read or write at runtime (*run-time properties*). (You can set some properties at both design time and runtime.)

To modify design time properties, you generally require a license from the control vendor. The vendor typically provides this license to application developers (as opposed to application end-users) for installation with the control.

The AppBuilder provides access to all available design time properties using the **OCX Property Editor** window. This window contains all design time properties including the extended properties that ABL adds. This is the only way to set values for properties writable only at design time.

To modify run-time properties, you write ABL code that sets the property value using the component handle to the control.

For more information on the available design time properties, see the list displayed in the **OCX Property Editor** window, and consult your ActiveX control documentation. For information on the available run-time properties, see the list displayed in the OpenEdge COM Object Viewer (see Using COM Objects in ABL on page 523), and consult your ActiveX control documentation.

Setting design time properties

Unlike ABL widgets, which have a standard set of defaults for the common attributes, the default settings for common ActiveX control properties vary from control to control. If you observe unexpected run-time behavior from an ActiveX control, consider whether you have set design time property values that are appropriate for your application.

Note: Similar considerations apply to setting run-time property values for ActiveX controls. For more information, see Programming ActiveX controls in the AppBuilder on page 570.

Setting the ActiveX control name

The AppBuilder creates a default name (OCX name) for the ActiveX control instance when you first insert it into the design window. (This is the name defined by the control vendor.) You can change this name in the AppBuilder through the design time Name property. The AppBuilder identifies the control instance by both the control-frame name and the OCX name, rather than the OCX name alone. Because the control-frame name is always unique, the control-frame/OCX name pair is always unique.

If you do decide to change the ActiveX control name, be aware that ABL uses this name to identify event handlers for the control. If you define event handlers in external procedures other than the one where you initially define the control instance, the AppBuilder cannot update them to conform with the new name. You must modify the names of these external event handlers manually to conform with the new name. For more information on defining event handlers for ActiveX controls, see Handling events on page 565.

Note: You should not change the control name at runtime. If you do, the event handlers for the control will not work, since an event handler name is formed in part from the name of the control-frame and the control.

Setting the control-frame name

When you first insert an ActiveX control into the design window, the AppBuilder creates a unique default Object name for the control-frame that contains the ActiveX control. As with any ABL widget, you can change this Object name in the AppBuilder. However, you must use extra care when changing this name. The AppBuilder uses the control-frame Object name for three program elements:

- As the value for the control-frame widget `NAME` attribute (and COM object Name property). When loading the associated ActiveX control at runtime, the AppBuilder locates the control instance in the section of the `.wrx` file identified by this name. When responding to ActiveX control events, ABL uses this name to identify event handlers for the control.

- As the variable name for the widget handle of the control-frame widget.
- As a part of the variable name (prefixed by "ch") for the component handle of the control-frame COM object.

When you change the control-frame Object name manually in the AppBuilder, the AppBuilder automatically updates these three program elements and all AppBuilder-generated code. When you save the application file, aside from saving the changes to these program elements, it also updates the corresponding section name in the `.wrx` file.

However when you change this Object name, the AppBuilder does not update any custom code you have added using the Section Editor. It also does not update the identity of any event handlers that you might have defined for the control in external procedures other than the one where you initially defined the control. You must update this code manually in the Section Editor or external procedure file. (For more information on how ABL identifies event handlers for ActiveX controls, see Handling events on page 565.)

Note: Progress Software Corporation does not recommend changing the control-frame name at runtime. If you do, the event handlers for the control will not work, since an event handler name is formed in part from the name of the control-frame and the control.

Orienting a control in the user interface at design time

The control-frame manages the location and dimensions of the associated ActiveX control in the user interface. Thus, to orient the ActiveX control at design time, you must set certain control-frame attributes. You can set these attributes directly, using the AppBuilder Property Sheet. You can also set them indirectly by visually manipulating the control in the design window.

Note: For ABL widgets, you can open the AppBuilder Property Sheet by double-clicking on the widget. However for ActiveX controls, this opens the **OCX Property Editor** window. To access the control-frame widgets Property Sheet, you must select the ActiveX control, then click **Object properties** or choose **Tools** > **Property Sheet**.

Setting control location

The AppBuilder automatically updates the control location wherever you insert or move it in the design window. This movement updates the `COLUMN`, `ROW`, `X`, and `Y` attributes of the control-frame widget (and the corresponding control-frame COM object properties). You can also modify these attributes manually in the Property Sheet of the control-frame.

Setting control height and width

The AppBuilder automatically updates the size of the control whenever you use the mouse to adjust the resize handles. These resize handles actually belong to the control-frame and the ActiveX control generally conforms to the dimensions you set for the control-frame.

There are some exceptions to this. Some controls depend on a specific size. For example, if the control is represented by an icon, it expects to be sized appropriately (for example, 32 pixels square). Another example is a combo box that expects its size to be only high enough to hold its text. For controls like these, if the user tries to resize the control-frame in a way that violates its internal sizing rules, the control automatically snaps back to the size that it requires. In this case, the control-frame resizes to fit the control rather than the other way around.

For resizable controls, these adjustments update the `HEIGHT-CHARS`, `HEIGHT-PIXELS`, `WIDTH-CHARS`, and `WIDTH-PIXELS` attributes of the control-frame widget (and the corresponding control-frame COM object properties). You can also modify these attributes manually in the Property Sheet of the control-frame.

Setting control tab order

Control tab order is established by the attributes of the control-frame widget. The control-frame is a dynamic widget parented (by the AppBuilder) to a static frame. As such, each control-frame in the frame assumes a default tab order that begins after any static input widgets (such as buttons) parented by the frame. However, the AppBuilder generates code that adjusts the tab order of widgets to match the order that you insert them in the design window. At design time, you can reset the tab order of widgets different from the insertion order using the **Tab Editor** dialog box in the AppBuilder. For information on managing the tab order of ActiveX controls at runtime, see Managing ActiveX controls at runtime on page 562.

Note: As you tab between ActiveX controls and field-level widgets at design time or at runtime, ABL shows no indication of focus in the ActiveX controls. Otherwise, ActiveX controls behave like any other widget in an ABL frame.

Defining invisible controls

For controls with a run-time user interface (such as the CSSpin control provided with OpenEdge), you can make them initially invisible by setting the Hidden option in the control-frame Property Sheet. (This sets the control-frame widget `HIDDEN` attribute to `TRUE`.)

For controls without a run-time user interface (such as the PSTimer control provided with OpenEdge), the AppBuilder automatically sets the control-frame `HIDDEN` attribute to `TRUE`. For such controls, you cannot change this attribute from the Property Sheet.

Also, for a control without a run-time user interface, location, size, and tab order generally have no run-time significance. At design time, however, you might want to position (and resize, if possible) the control's design time representation for ease of maintenance.

Note: ABL automatically removes some controls from the tab order if they have a flag indicating that they should not receive input focus. These controls might be visible. For example, the Status bar control is visible, but never gets focus.

Accessing ActiveX controls at runtime

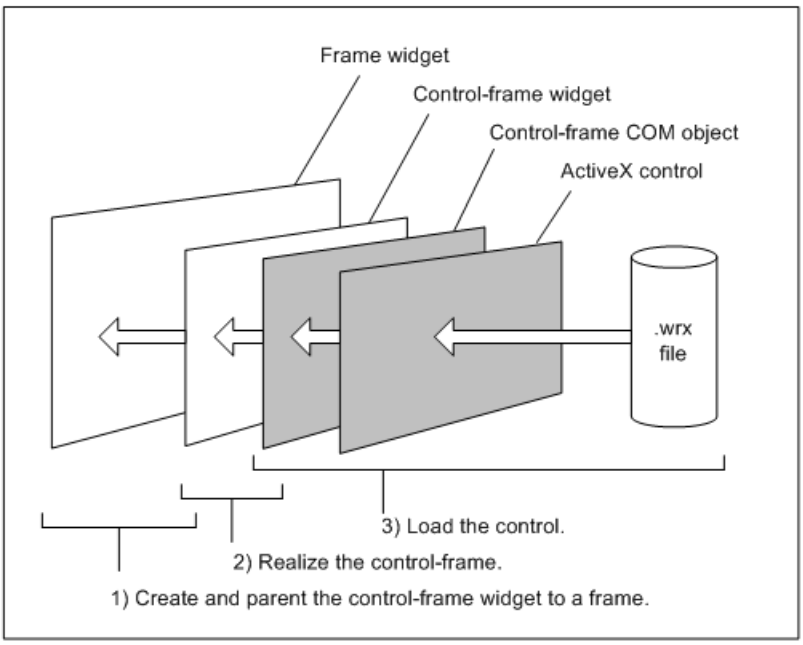
To access an ActiveX control at runtime, your application must instantiate the control and get the component handle to the ActiveX control.

Instantiating the control

At runtime, your application uses AppBuilder-generated code to instantiate any ActiveX controls. This code (shown in following sections) uses the `CREATE Widget` statement to realize a separate control-frame widget for each ActiveX control, initializing each control-frame with the name specified at design time. It then invokes the `LoadControls()` method on each control-frame COM object to instantiate the corresponding ActiveX control, loading all design time definitions from the `.wrx` file.

The following figure describes the instantiation process for a single ActiveX control.

Figure 60: Instantiating an ActiveX control at runtime



Creating the control-frame widget does not, by itself, realize the control-frame. Control-frame realization occurs only after (1) it is parented to a frame widget and (2) its `NAME` attribute is set or its `COM-HANDLE` attribute is referenced. The `NAME` attribute is generally set first and causes realization by creating the control-frame COM object. Only then can your application use the `LoadControls()` method to (3) load the control into the control-frame COM object (the control container).

Note: The control-frame name must match the name specified at design time to allow `LoadControls()` to locate the specified ActiveX control in the `.wrx` file.

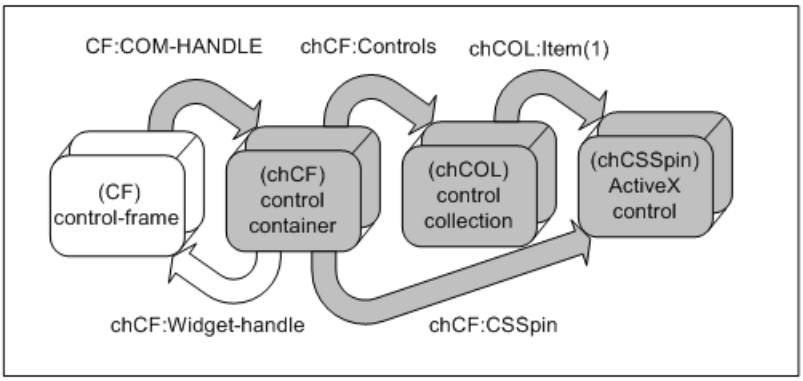
Accessing the control handle

Once the ActiveX control is instantiated, the control-frame affords access to one widget and three COM objects, including the:

- Control-frame widget
- Control container (the control-frame COM object)
- Control collection (a standard COM object)
- ActiveX control (the target COM object)

The following figure shows the chain of handle references that connect a control-frame widget (with widget handle CF) to an ActiveX control (with the Name CSSpin).

Figure 61: Handle references to access ActiveX controls



The chCF reference is a component handle to the control-frame COM object (control container) and the chCOL reference is a component handle to the control collection. Thus in ABL, you can instantiate and get the handle to an ActiveX control in two ways:

- Using the name of the ActiveX control (CSSpin) as if it were a property (*Control-Name* property) of the control container (chCF:CSSpin)
- Using the indexed **Item** property of the control collection (chCOL:Item(1))

Once you have the component handle of the ActiveX control, you can access all of its properties and methods. (The chCSSpin reference is a component handle variable you might set from chCF:CSSpin or chCOL:Item(1).) The following examples reference a reduction of typical code generated by the AppBuilder. For a closer look at actual code generated by the AppBuilder, see Programming ActiveX controls in the AppBuilder on page 570.

Using the Control-Name property of the control container

This example show how you might access the component handle to the CSSpin control using the *Control-Name* property:

```
DEFINE VARIABLE chCSSpin    AS COM-HANDLE. /* ActiveX Control */
DEFINE VARIABLE CtrlFrame   AS HANDLE.     /* Control-Frame Widget */
DEFINE VARIABLE chCtrlFrame AS COM-HANDLE. /* Control-Frame COM Object */
DEFINE VARIABLE OCXFile     AS CHARACTER   /* .wrx file pathname */

/* Define a FRAME widget named 'Foo' */

/* Create OCX Container */

CREATE CONTROL-FRAME CtrlFrame ASSIGN  /
  FRAME = FRAME Foo:HANDLE.
CtrlFrame:NAME = "CtrlFrame":U.

/* Load (instantiate) ActiveX control */

OCXFile     = SEARCH ( "csspinapp.wrx":U ).
chCtrlFrame = CtrlFrame:COM-HANDLE.
chCtrlFrame:LoadControls( OCXFile, "CtrlFrame":U ).

chCSSpin = chCtrlFrame:CSSpin /* Control handle via the Control Name */
```

The bolded code is the code you might add to appropriate sections of your application; the rest is a simplified version of code the AppBuilder might generate. Note how the AppBuilder uses the control-frame name (`CtrlFrame`) to generate handle variable names and to locate the control instance in the `csspinapp.wrx` file. The actual control name (`CSSpin`) is the OCX name specified for the control at design time (in this case, the `CSSpin` default).

Using the item property of the control collection

This similar example uses the control collection to return the control's component handle:

```
DEFINE VARIABLE chCSSpin AS COM-HANDLE.  /* ActiveX Control */

/* The AppBuilder-generated code from the previous example */

chCSSpin = chCtrlFrame:Controls:Item(1). /* Control handle via Item( ) */
```

The handle connections

The previous examples rely on attributes, properties, and methods supported by ABL to get a component handle from a control-frame widget handle (see Figure 61: Handle references to access ActiveX controls on page 561). Thus, to provide access to the control-frame widget and its related COM objects:

- The control-frame widget has a `COM-HANDLE` attribute that returns the component handle of the control-frame COM object.
- The control-frame COM object has a Controls property that returns the component handle of the control collection. You can then use the **Item** property on the collection component handle to get the component handle of the control.

  Note: The control collection is a standard COM object that is used to access sets of like objects (objects in the same class).

- The control-frame COM object has a special property named after the ActiveX control that returns the component handle of the control.
- The control-frame COM object provides a Widget-Handle property that returns the widget handle of the instantiating control-frame widget.

Managing ActiveX controls at runtime

At runtime, the control-frame itself is largely irrelevant from the end-user viewpoint. It is the ActiveX control itself that is visible and useful. In general, the control exhibits its standard behavior as if the control-frame were not there.

The control does not follow any special ABL rules except those described in the following sections. For example, edit controls do not have any of the customized ABL text editing behavior. This means that they do not abide by ABL formats or dictionary validation rules. They are not affected by any OpenEdge startup parameters including, for example, `-d` (Date Format), `-E` (European Numeric Format), and -yy (Century). You can make any format, validation, or international settings using either the control's own properties or by changing regional settings in the system control panel. You can also write ABL code to do formatting and validation for a control.

However, ABL does enforce some rules and provides ABL mechanisms that allow ActiveX controls to live more comfortably with other ABL widgets in an application.

Managing tab order and Z order

Like any other ABL widget, ActiveX controls participate in the tab order of the ABL frame to which they are parented. The tab order of a control is managed by the control-frame widget. You can modify the tab order of a control using the control-frame's MOVE-AFTER-TAB-ITEM( ) and MOVE-BEFORE-TAB-ITEM( ) widget methods.

The same is true of Z order (the overlay order of controls). You can modify the Z order of a control using the control-frame's MOVE-TO-BOTTOM( ) and MOVE-TO-TOP( ) widget methods.

Working with ABL key functions

Setting the HonorProKeys property to TRUE allows the user to specify that ABL should handle certain key functions.

TAB, BACK-TAB, GO, HELP, and END-ERROR

ABL enforces the TAB and BACK-TAB key functions in all ActiveX controls. If a control defines a meaning for the **TAB** or **SHIFT+TAB** key other than normal tabbing, the control-defined function will not work. (Few, if any, controls redefine the **TAB** or **SHIFT+TAB** key in this way.)

ABL also enforces behavior on the three keys that implement the GO, HELP and END-ERROR functions (normally the **F2**, **F1** and **ESC** keys, respectively). Again, if the control normally uses these keys for another purpose, that functionality is lost in ABL. Note that the standard use of the **F1** and **ESC** keys in Windows applications matches the standard use of HELP and END-ERROR in ABL, and ActiveX controls do not generally use **F2**.

All three of these key functions work if focus is in a subwindow of an ActiveX control. For example, suppose a calendar control has two subwindows where one holds the month and the other holds the year. If you click into one of these subwindows and press **TAB**, focus moves to the next control or widget in the frame.

To allow the control to process the **GO**, **HELP**, **TAB**, and **ENDKEY** keys, set the HonorProKeys extended property to FALSE.

RETURN and Default Buttons

In Windows, the RETURN key function (normally the **ENTER** key) has special significance in that it can invoke the default button in a dialog box. You can program this functionality in ABL by designating a button as the DEFAULT-BUTTON of a frame. However, because it is common for an ActiveX control to specify its own use for the **ENTER** key, ABL does not trap this key in ActiveX controls. Therefore, RETURN does not activate the default button in an ABL frame if an ActiveX control has focus.

To override this behavior, set the HonorReturnKeys property to TRUE, so ABL can handle the **RETURN** key.

Setting graphical properties of an ActiveX control

In ABL you can set graphical properties of an ActiveX control with the LOAD-PICTURE statement. The LOAD-PICTURE statement takes a filename of a graphical object and returns a COM-HANDLE to an OlePictureObject. For example, the Microsoft Image control has a picture property that controls the image displayed in the control. The type of this property is equivalent to an ABL COM-HANDLE.

Releasing control resources

The following material describes the process of creating and freeing control-frame widgets. This is automatically done by the AppBuilder, but, like any other dynamic widget, you can delete a control-frame some time after you create it.

You can delete a control-frame using two techniques:

- Associate the control-frame with a widget pool in the CREATE statement. When you delete the widget pool, ABL deletes the control-frame widget and also releases its control-frame COM object. This is the default technique used by the AppBuilder. (The AppBuilder uses the default unnamed widget pool that is deleted when the .w ends.)
- Explicitly delete the control-frame using the DELETE WIDGET statement. This statement deletes the control-frame widget and also releases the control-frame COM object.

Releasing ActiveX controls

When you delete a control-frame widget, ABL also automatically releases the control-frame COM object as well as any references to the ActiveX control held by the control-frame. You must release all other COM objects using the RELEASE OBJECT statement:

```
DEFINE VARIABLE CtrlFrame    AS HANDLE     NO-UNDO.
DEFINE VARIABLE chCtrlFrame  AS COM-HANDLE NO-UNDO.
DEFINE VARIABLE chCSSpin     AS COM-HANDLE NO-UNDO.
DEFINE VARIABLE chCollection AS COM-HANDLE NO-UNDO.

/* Create frame Foo and instantiate control */
CREATE CONTROL-FRAME CtrlFrame
  ASSIGN
    FRAME = FRAME Foo:HANDLE
    NAME  = "CtrlFrame":U.

chCtrlFrame = CtrlFrame:COM-HANDLE.
chCtrlFrame:LoadControls("csspinapp.wrx":U, "CtrlFrame":U).

chCollection        = chCtrlFrame:controls.
chCSSpin            = chCollection:Item(1).
chCSSpin:ShadeColor = RGB-VALUE(0,128,0).
RELEASE OBJECT chCollection.

/* Do some more stuff ... WAIT-FOR ... */

DELETE WIDGET CtrlFrame.
```

This example releases the control collection after it is no longer needed. It also deletes the control-frame using the DELETE WIDGET statement, which also releases the chCtrlFrame COM object as well as the ActiveX control itself (chCSSpin).

Releasing COM objects individually

If you try to release an ActiveX control (using the RELEASE OBJECT statement) before the control-frame is deleted, this works but is unnecessary. If you try to access a control after the control-frame is deleted, the ABL Virtual Machine (AVM) displays an error message that you are trying to reference an invalid component handle.

Because you have a component handle to a control-frame COM object, you might think you can release it using the RELEASE OBJECT statement. However for control-frame COM objects, ABL does not allow this because of the link between the control-frame widget and COM object.

Thus, you can only release the control-frame COM object by deleting the control-frame widget. If you do try to release the component handle of a control-frame, the AVM returns an error indicating that you should delete the object through the widget handle instead.

In general, if you do not delete or release any COM objects in an application, all active COM objects remain instantiated until the end of the OpenEdge session, at which time the AVM automatically releases them.

For more information on releasing COM object resources, see Using COM Objects in ABL on page 523.

Handling events

ABL processes two types of events for an ActiveX control:

- **Field-level widget events on the control-frame** — A subset of standard ABL events
- **ActiveX control events** — A set of events that are unique to each ActiveX control and that often pass parameters

The requirements for these two types of events are different. Thus, ABL uses separate mechanisms to handle events on the control-frame as opposed to events received from the associated ActiveX control. In addition, the two types of events are mutually exclusive and any action on the control generates either one type of event or the other.

Handling control-frame widget events

You can handle field-level widget events for the control-frame like ABL events for any other dynamic ABL widget, with ABL triggers built from the `ON` statement or the *`trigger-phrase`* of the `CREATE` statement.

The control-frame supports a subset of field-level widget events that ABL requires for:

- **Managing application execution** — Universal key function events: `END-ERROR` (which occurs when the `END` key is pressed), `GO`, and `HELP`
- **Tabbing between the control and other field-level widgets** — Navigation key function events: `TAB` and `BACK-TAB`
- **Detecting any change of focus** — High-level widget events: `ENTRY` and `LEAVE`
- **Handling programmer-defined events** — Developer events: `U1` through `U10`

Note: Unlike earlier support for VBX controls, you do not have to handle key function events as KeyPress events. With VBX controls, you had to trap key functions as KeyPress events in a VBX control event handler, then apply them to the VBX control-container widget to handle with corresponding key function event triggers. With ActiveX controls, your triggers on the control-frame widget now handle the supported function events automatically.

Handling ActiveX control events

Unlike ABL widget events, ActiveX control events often pass parameters. Also, while there are a few standard events common to most ActiveX controls, there are an infinite variety of possible events that are unique to each ActiveX control. Because ABL cannot have direct knowledge of all these possible events and must be able to handle the parameters of many of them, the standard ABL trigger mechanism cannot handle them.

Instead, ABL allows you to handle ActiveX control events using OCX event procedures. An *OCX event procedure* is a standard ABL internal procedure that serves as an event handler for ActiveX controls. The parameter-passing mechanism provided for ABL procedures handles most parameters that ActiveX control events typically pass. ABL identifies an OCX event procedure from the way its name is put together. This is the only syntactic feature that distinguishes the ABL internal procedure as an OCX event procedure.

Creating event procedures in the AppBuilder

In the AppBuilder, you define OCX event procedures as OCX event triggers. You can identify each ActiveX control event in the AppBuilder event list by the "OCX." prefix attached to the event name. When you select an event name to define an OCX event trigger, the AppBuilder automatically generates an internal procedure block for the event procedure, including any procedure parameter definitions.

Coding event procedures

When you create an OCX event trigger in the AppBuilder, the AppBuilder defines an internal procedure template with the following components:

- A procedure name that identifies the ActiveX control and event being handled
- Any required procedure parameter definitions with default mode and data type mappings

If you code OCX event procedures in external procedure files other than the one where you instantiate the ActiveX control, you must code the complete event procedure yourself. In addition, you must also add the procedure to the list that ABL searches to find event procedures to handle events. For more information, see Managing external procedure files on page 569.

Coding event procedure names

ABL supports two types of OCX event procedures, distinguished by the procedure name:

- Control-bound event procedures that handle a specific event for a specific control instance
- Generic event procedures that handle a specific event for all ActiveX controls in the application

---

Note: Names of event procedures are not case sensitive.

The names for control-bound event procedures contain three parts, delimited by a period (.). Each part can be quoted if it contains embedded spaces:

1. The name of the control-frame (the NAME attribute value of the control-frame widget
2. The name of the ActiveX control (the Name property value set in the Property Editor)
3. The name of the event that is handled by the procedure

As with all internal procedures that you add in the AppBuilder, the PROCEDURE statement and procedure name is not visible in the Section Editor. However, this is one that you or the AppBuilder might code:

```
PROCEDURE CtrlFrame.CSSpin.SpinUp .
```

This begins the definition for a procedure to handle the SpinUp event for the control named CSSpin in the control-frame named CtrlFrame.

The names for generic event procedures contain two parts, delimited by a period (.):

1. ANYWHERE
2. The name of the event that is handled by the procedure

The AppBuilder does not provide a mechanism to generate generic event procedures. You must code these yourself in the AppBuilder using the **New Procedure** dialog box, or you can add them to an external procedure file. For example:

```
PROCEDURE ANYWHERE.SpinUp .
```

A generic event procedure with this name handles the SpinUp event for any control instance that does not have a control-bound event procedure defined for the SpinUp event.

Coding event parameter definitions

As for any procedure definition, you code event procedure parameters using the `DEFINE PARAMETER` statement. When you create an event procedure as an OCX trigger in the AppBuilder, the AppBuilder provides default definitions for all required parameters. These default definitions attempt to specify an ABL data type that is compatible with the COM data type of the corresponding OCX event parameter. This data type generally follows the data type mappings shown in the following table.

Table 64: Data type mappings for OCX event parameters

COM data type	ABL data type
Array	ABL array variable
Array of bytes	`RAW`
Boolean	`LOGICAL`
Byte	`INTEGER`
Currency	`DECIMAL`
Date Double	**DATE** **DATETIME** **DATETIME-TZ**
Double	`DECIMAL`
Integer (2-byte integer)	`INTEGER`
Long (4-byte integer)	`INTEGER`
Object (COM)	`COM-HANDLE`
Single (float)	`DECIMAL`
String	`CHARACTER LONGCHAR`
Unsigned byte	`INTEGER`
Unsigned long (4-byte integer)	`INTEGER`

COM data type	ABL data type
Unsigned short (2-byte integer)	INTEGER
Variant	*ANYTYPE*[11]
VT-ARRAY (if single-dimensional array of bytes)	RAW

For event procedures you write yourself, you can also find the suggested ABL data type for each parameter displayed in the OpenEdge COM Object Viewer for each selected control event. For more information on this viewer, see Using COM Objects in ABL on page 523.

In the AppBuilder, ABL uses data type mappings from the COM data type specified in the Type Library for the ActiveX control. If ABL cannot determine a data type mapping, the AppBuilder specifies *ANYTYPE* as a place holder. You must change *ANYTYPE* to the data type that most closely matches the expected value. ABL does its best to convert the COM data type to the ABL data type you specify. For more information on a COM data type, see the available documentation on the Microsoft Component Object Model and the event parameter you want to convert, see COM Object Data Type Mapping on page 609. For information on Type Libraries and ABL, see Using COM Objects in ABL on page 523.

The AppBuilder also provides a default mode option (INPUT, OUTPUT, or INPUT-OUTPUT) for each parameter definition. Here, although ABL does its best to interpret the information in the Type Library for the ActiveX control, this is not a direct mapping. You might have to modify the mode option. If the option chosen by ABL is incorrect, this generates a run-time error. If the event parameter is passed as read-only, define it as an INPUT parameter. If you can change its initial value and must pass it back to the ActiveX control, define it as an INPUT-OUTPUT parameter. If only the value you return to the ActiveX control has any meaning, define it as an OUTPUT parameter.

Caution: If an INPUT parameter passes in a COM-HANDLE value, use the RELEASE OBJECT statement to release the specified COM object when you no longer need it (usually before the event procedure returns). Otherwise, the component handle might become unavailable and the COM object might never be released. Also, not releasing the COM object in the event procedure can cause unpredictable behavior with some controls.

Caution: In most contexts, OCX event procedures behave as expected. However, if an event procedure has any INPUT-OUTPUT or OUTPUT parameters, and the event procedure executes while blocking on non-modal .NET forms, if the procedure directly or indirectly executes an additional ABL input-blocking or event processing statement (such as UPDATE or WAIT-FOR) that blocks on a non-modal ABL window, the output parameters are **not** returned to the control. For more information about restrictions when input-blocking on both .NET forms and ABL windows, see *Use GUI for .NET Objects in ABL Applications*.

Using system handles in OCX event procedures

ABL allows you to obtain the identities of both the ActiveX control that generates an event and the associated control-frame widget from within an OCX event procedure. The COM-SELF system handle returns the component handle of the ActiveX control and the SELF system handle returns the widget handle of the control-frame. This is especially useful in generic event procedures where the source of an event cannot be known.

Unlike for ABL ON triggers, the RETURN NO-APPLY statement has no affect in an OCX event procedure. Some COM object events provide a similar mechanism using output parameters. For example, the standard KeyPress event passes a key code as an INPUT-OUTPUT parameter. If you set this parameter to 0 in the event handler, the key is discarded.

[11] You must replace *ANYTYPE*, provided by the AppBuilder in some OCX event trigger templates, with a valid ABL data type.

Note: To trigger an ABL widget event from ABL, you use the APPLY statement. However, to trigger an ActiveX control event in ABL, you execute its OCX event procedure as a standard internal procedure, passing any required parameters.

Event handler example

The following example shows an OCX event trigger for the NewItem event of the CSComboBox control MyCombo in control-frame CtrlFrame-2:

```
PROCEDURE CtrlFrame-2.MyCombo.NewItem:
  DEFINE INPUT-OUTPUT PARAMETER i-ComboText AS CHARACTER NO-UNDO.
  DEFINE INPUT-OUTPUT PARAMETER i-KeepFocus AS INTEGER   NO-UNDO.

  COM-SELF:AddItem(i-ComboText, 1).
  i-KeepFocus = -1.
END PROCEDURE.
```

This event occurs when a user enters an item in the text box that is not in the combo box list.

The programmer uses the COM-SELF handle to add the new text item to the list with the AddItem() method. Setting i-KeepFocus to -1 ensures that the control maintains input focus after the item is added to the list.

If this event procedure is in the AppBuilder-generated application file, the programmer might otherwise choose to call the AddItem() method indirectly using the component handles provided by the AppBuilder (chCtrlFrame-2) and the control-frame COM object (MyCombo property):

```
PROCEDURE CtrlFrame-2.MyCombo.NewItem:
  DEFINE INPUT-OUTPUT PARAMETER i-ComboText AS CHARACTER NO-UNDO.
  DEFINE INPUT-OUTPUT PARAMETER i-KeepFocus AS INTEGER   NO-UNDO.

  chCtrlFrame-2:MyCombo:AddItem(i-ComboText, 1).
  i-KeepFocus = -1.
END PROCEDURE.
```

Managing external procedure files

As described earlier (see Handling ActiveX control events on page 565), you can define OCX event procedures in external procedure files other than the one where you instantiate the ActiveX control. If you do create these external procedure files, you need to tell ABL how to locate the event procedures when responding to ActiveX control events. You do this by creating a search list of your external procedures.

The control-frame widget provides two methods for you to manage this search list:

- ADD-EVENTS-PROCEDURE(*procedure-handle*)
- REMOVE-EVENTS-PROCEDURE(*procedure-handle*)

These methods require that your external procedures are running persistently or on the procedure call stack. To have ABL locate an event procedure in one of these external procedures, you invoke the ADD-EVENTS-PROCEDURE( ) on the control-frame widget, passing the procedure handle of the external procedure. To remove an external procedure from the search list, you pass its procedure handle to the REMOVE-EVENTS-PROCEDURE( ) method on the control-frame widget.

When ABL responds to a control event, it searches the assembled list of external procedures to find a matching control-bound event handler, searching in order of the most recently added procedure first. Finally, it searches the application file where the control is instantiated. If it cannot find a control-bound event handler, it searches the list again for a matching generic event handler, always searching the main application file last. In this way, ABL selects the first matching event handler to handle the incoming event.

Using these methods, you can dynamically add and remove procedures from the search list, overriding or replacing previously added procedures. Thus, modifying this search list can have a powerful effect on the behavior of an application.

Programming ActiveX controls in the AppBuilder

The code fragments that follow illustrate selected sections of the sample ActiveX control application, `i-ocx1.w`, shown running in Figure 58: ActiveX control example on page 552. Recall that this application uses a spin button control to scan `Customer` records in the `sports2000` database.

Creating data definitions for an ActiveX control

These fragments provide data and code definitions, including those for the spin button control, referenced by other sections of the application. The first fragment contains variable definitions that are hand-coded in the Definitions section of the AppBuilder Section Editor:

```
/* Local Variable Definitions ---                                       */

DEFINE VARIABLE result      AS LOGICAL    NO-UNDO.
DEFINE VARIABLE max-records AS INTEGER    NO-UNDO.
DEFINE VARIABLE chCSSpin    AS COM-HANDLE NO-UNDO.
```

Note the variable to hold the component handle CSSpin ActiveX control. All other necessary widget and component handles for this application are provided by the AppBuilder.

This is the query definition assembled by the AppBuilder from user input. The `PRESELECT` option allows the application to know the number of records it is scanning with the spin button at startup, as shown:

```
/* ********************  Preprocessor Definitions  ******************** */
. . .

&SCOPED-DEFINE OPEN-QUERY-Dialog-Frame OPEN QUERY Dialog-Frame PRESELECT EACH
sports2000.Customer NO-LOCK.
```

The following code section shows relevant widget handle, component handle, and query definitions generated by the AppBuilder from user input:

```
/* ***********************  Control Definitions  ********************** */
. . .
/* Definitions of handles for OCX Containers                            */
DEFINE VARIABLE custSpin   AS HANDLE     NO-UNDO.
DEFINE VARIABLE chcustSpin AS COM-HANDLE NO-UNDO.
. . .

DEFINE VARIABLE iRecordCount AS INTEGER FORMAT "9999":U LABEL "Record Number"
  VIEW-AS TEXT SIZE 5.8 BY .62 NO-UNDO.
. . .

/* Query definitions                                                    */
. . .
DEFINE QUERY Dialog-Frame FOR sports2000.Customer SCROLLING.
```

The AppBuilder assembles the widget definitions as you add objects to the design window. Thus, the AppBuilder assembles the definitions for the control-frame handles (`custSpin` and `chcustSpin`) after you insert the spin button control into the design window. (The programmer has changed the name of the control-frame from `CtrlFrame` to `custSpin`.) The application also uses the `iRecordCount` variable to display the ordinal record number of each **Customer** record scanned by the spin button control.

Note:

You can begin setting ActiveX control properties in the OCX Property Editor Window any time after you add the control into the design window.

You cannot view or change this AppBuilder-generated code from the Section Editor. However, it is part of the `.w` file generated for the application, and you can use Code **Preview** option of the **Tools** menu to view this code.

The AppBuilder automatically generates the query definition from the query criteria entered earlier.

This fragment shows the actual control-frame definition created by the AppBuilder using the `custSpin` widget handle defined earlier. The AppBuilder creates and maintains this definition as you insert, position, and size the spin button control in the design window. Note the setting of the control-frame `NAME` attribute after the widget is created and parented to the Dialog-Frame frame. The AppBuilder also sets the `HIDDEN` attribute to no (`FALSE`) because the spin button control is, by default, a visible control. For example:

```
/* **********************  Create OCX Containers  ********************** */

&ANALYZE-SUSPEND _CREATE-DYNAMIC

&IF "{&OPSYS}" = "WIN32":U AND "{&WINDOW-SYSTEM}" NE "TTY":U &THEN
CREATE CONTROL-FRAME custSpin ASSIGN
       FRAME        = FRAME Dialog-Frame:HANDLE
       ROW          = 11.19
       COLUMN       = 15.4
       HEIGHT-CHARS = 2.33
       WIDTH-CHARS  = 61.2
       HIDDEN       = no
       SENSITIVE    = yes.
      custSpin:NAME = "custSpin":U .
/* custSpin OCXINFO:CREATE-CONTROL from:
{EAF26C8F-9586-101B-9306-0020AF234C9D} type: CSSpin */
      custSpin:MOVE-AFTER(Btn_Update:HANDLE IN FRAME Dialog-Frame).
&ENDIF

&ANALYZE-RESUME /* End of _CREATE-DYNAMIC */
```

Note: You cannot view or change this AppBuilder-generated code from the Section Editor. However, it is part of the `.w` file generated for the application, and you can use Code **Preview** option of the **Tools** menu to view this code.

The AppBuilder also ensures, with appropriate preprocessor settings, that ABL only compiles and executes OCX-related sections of this procedure file if it is running in the graphical mode (not TTY) of a Windows system (`WIN32`).

Instantiating the control

The AppBuilder generates the code to instantiate the ActiveX control starting with a call to the `enable_UI` procedure from the Main Block. For example:

```
/* ***************************  Main Block  *************************** */

/* Parent the dialog-box to the ACTIVE-WINDOW, if there is no parent.   */
IF VALID-HANDLE(ACTIVE-WINDOW) AND FRAME {&FRAME-NAME}:PARENT EQ ? THEN
  FRAME {&FRAME-NAME}:PARENT = ACTIVE-WINDOW.

/* Now enable the interface and wait for the exit condition.            */
/* (NOTE: handle ERROR and END-KEY so cleanup code will always fire.    */
MAIN-BLOCK:
DO ON ERROR   UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK
   ON END-KEY UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK:
  RUN enable_UI.

  /* Set max-records from open query in enable_UI. */
  max-records = NUM-RESULTS("Dialog-Frame").

  WAIT-FOR GO OF FRAME {&FRAME-NAME}.
END.
RUN disable_UI.
```

The programmer has added the comments and code that immediately follow the call to `enable_UI`. Here, they must set max-records to the number of records in the query only after the query is opened (again, in `enable_UI`), but before the spin button control is available to scan the result. The reason for this becomes clearer in later code fragments.

The following reduction of the `enable_UI` procedure shows that the AppBuilder generates the code to load the ActiveX control (`RUN control_load`) before its static parent frame and family of ABL widgets are displayed and enabled:

```
/* **********************  Internal Procedures  *********************** */
. . .

PROCEDURE enable_UI :
/*------------------------------------------------------------------------------
  Purpose:     ENABLE the User Interface
  Parameters:  <none>
  Notes:       Here we display/view/enable the widgets in the
               user-interface.  In addition, OPEN all queries
               associated with each FRAME and BROWSE.
               These statements here are based on the "Other
               Settings" section of the widget Property Sheets.
------------------------------------------------------------------------------*/
  RUN control_load.

  {&OPEN-QUERY-Dialog-Frame}
  GET FIRST Dialog-Frame.
  DISPLAY iRecordCount
    WITH FRAME Dialog-Frame.
  . . .

END PROCEDURE.
```

The initial value of `iRecordCount` is also set at control load time (see Initializing the control on page 575). Note also that the application query is opened to obtain the data for the frame:

Note: You can view this code in the Procedures section of the AppBuilder Section Editor. However, unlike the Main Block, this is protected AppBuilder-generated code that you cannot change.

The control_load procedure, called from enable_UI, is an AppBuilder-generated procedure that loads the ActiveX control into the control-frame by executing the LoadControls( ) method of the control-frame COM object. Note the call to initialize-controls, an optional internal procedure that you can define (and which is defined in this example) to modify control properties before the control becomes visible. For example:

```
/* **********************  Internal Procedures  ********************** */
. . .

PROCEDURE control_load :
/*------------------------------------------------------------------------
  Purpose:     Load the OCXs
  Parameters:  <none>
  Notes:       Here we load, initialize and make visible the
               OCXs in the interface.
------------------------------------------------------------------------*/
&IF "{&OPSYS}" = "WIN32":U AND "{&WINDOW-SYSTEM}" NE "TTY":U &THEN
DEFINE VARIABLE UIB_S    AS LOGICAL    NO-UNDO.
DEFINE VARIABLE OCXFile  AS CHARACTER  NO-UNDO.

OCXFile = SEARCH( "i-ocx1.wrx":U ).

IF OCXFile <> ? THEN DO:
  ASSIGN
    chcustSpin = custSpin:COM-HANDLE
    UIB_S = chcustSpin:LoadControls( OCXFile, "custSpin":U).
  RUN initialize-controls IN THIS-PROCEDURE NO-ERROR.
END.
ELSE MESSAGE "The file, i-ocx1.wrx, could not be found." skip
             "The controls cannot be loaded."
             VIEW-AS ALERT-BOX TITLE "Controls Not Loaded".
&ENDIF
END PROCEDURE.
```

Note:

You can view but you cannot change this AppBuilder-generated code from the Section Editor.

You might wonder why you cannot load the control with a chained handle reference CtrlFrame:COM-HANDLE:LoadControls(...). The reason is that even though CtrlFrame:COM-HANDLE returns a component handle, it does so with reference to a widget attribute (COM-HANDLE), not a COM object property or method. You cannot make a component handle expression by chaining widget handle references (that return component handles) with component handle references.

Initializing the control

The programmer creates the initialize-controls procedure if they want to modify control properties before the control becomes visible. In addition to initializing the component handle to the CSSpin control, the example procedure gets the initial setting of the CSSpin Value property. This allows the correct value to display when `enable_UI` visualizes the **Customer Information** dialog box for the first time. For example:

```
/* **********************  Internal Procedures  ********************** */
. . .

PROCEDURE initialize-controls :
/*------------------------------------------------------------------------
  Purpose:
  Parameters:  <none>
  Notes:
------------------------------------------------------------------------*/
  chCSSpin = chcustSpin:CSSpin.

  /* Initialize iRecordCount from the initial spin control value after
     control is loaded. */
  iRecordCount = chCSSpin:Value.
END PROCEDURE.
```

Note: The initialize-controls procedure is the earliest place provided by the AppBuilder Section Editor to dynamically modify properties for UI-enabled ActiveX controls.

After an ActiveX control instance is loaded, you can interact with it in several ways. You have just seen how you might retrieve a property value to display during user interface initialization. The most common interactions with an ActiveX control occur in the ABL event procedures that you write to handle OCX events.

Using event procedures

The AppBuilder provides definition templates for event procedures through the Triggers section of the Section Editor. (For more information, see Handling events on page 565.) In this application, the CSSpin control's SpinUp and SpinDown events are caught to advance or backup in the list of **Customer** records. These events also increment (SpinUp) or decrement (SpinDown) the spin button control Value property, which the application displays in the `iRecordCount` field.

This is the event procedure for the SpinUp event:

```
/* ************************  Control Triggers  ************************ */
. . .

PROCEDURE custSpin.CSSpin.SPINUP .
/*-------------------------------------------------------------------------
  Purpose:
  Parameters Required for this OCX:
   None
  Notes:
-------------------------------------------------------------------------*/
  GET NEXT Dialog-Frame.
  IF NOT AVAILABLE Customer THEN DO:
    /* Control is incremented to last Customer + 1 */

    max-records = chCSSpin:Value - 1.
    chCSSpin:Value = 1.
    GET FIRST Dialog-Frame.
  END.
  RUN displayCustomer.
END PROCEDURE.
```

As you hold down the right-arrow button of the control, the control generates continuous SpinUp events. For each such event, this procedure reads the next query record and displays it. However, if the procedure tries to read beyond the last record in the query, it resets max-records to 1 less than the current Value property setting (the last record count).

It also resets the Value property to 1, wrapping around and restarting the upward query scan at the first record. Thus, the user can scan upward continuously through the query without reversing direction.

Note: The reason the application calculates a new value for max-records when it passes the last record is to get an updated result if records are deleted after `NUM-RESULTS` from the query is initially returned. However, note that the value might be inaccurate if records are deleted after the user encounters them. So, this is not a perfect solution.

This is the event procedure for the SpinDown event. As you hold down the left-arrow button of the control, the control generates continuous SpinDown events. For each such event, this procedure reads the previous query record and displays it. However, if the procedure tries to read before the first record in the query, it resets the current Value property setting to the known count of the last record in the query and restarts the downward query scan at the last record. For example:

```
/* ************************  Control Triggers  ************************ */
. . .

PROCEDURE custSpin.CSSpin.SPINDOWN .
/*-------------------------------------------------------------------------
  Purpose:
  Parameters Required for this OCX:
   None
  Notes:
-------------------------------------------------------------------------*/
  GET PREV Dialog-Frame.
  IF NOT AVAILABLE Customer THEN DO:
    chCSSpin:Value = max-records.
    GET LAST Dialog-Frame.
  END.
  RUN displayCustomer.
END PROCEDURE.
```

Thus, the user can scan downward continuously through the query without reversing direction.

Note: The reason the application needs to set max-records from the number of query entries at the very start of execution (in the Main Block) might already be apparent. If the user begins by scanning downward from the first record, the application now knows the correct value to reset the record counter (Value property) to start the downward query scan at the last record.

Interacting outside of event procedures

Finally, this is a simple example of where you might reference an ActiveX control from a procedure other than an event procedure:

```
/* **********************  Internal Procedures  ********************** */
. . .

PROCEDURE displayCustomer :
/*------------------------------------------------------------------------
  Purpose:
  Parameters:  <none>
  Notes:
------------------------------------------------------------------------*/

  iRecordCount = chCSSpin:Value.
  DISPLAY sports2000.Customer.Name sports2000.Customer.Address
    sports2000.Customer.Address2 sports2000.Customer.Balance
    sports2000.Customer.City sports2000.Customer.Comments
    sports2000.Customer.Contact sports2000.Customer.Country
    sports2000.Customer.CreditLimit sports2000.Customer.CustNum
    sports2000.Customer.Discount sports2000.Customer.PostalCode
    sports2000.Customer.SalesRep sports2000.Customer.Terms
    sports2000.Customer.State sports2000.Customer.Phone
    iRecordCount
    WITH FRAME Dialog-Frame.
END PROCEDURE.
```

The `displayCustomer` procedure retrieves a changed setting of the CSSpin Value property and displays it as the current record position for any event that changes that position. Other interactions are possible within triggers for control-frame events and by passing the control handle to external and persistent procedures.

ActiveX controls and examples installed with ABL

OpenEdge comes installed with three ActiveX controls, all of which are licensed for design mode. The following sections provide a brief overview of what each control does. For complete information on each control and its supported properties, methods, and events, see the OpenEdge Online Reference that comes with your installation or access online help from any tool of the OpenEdge Application Development Environment (ADE).

Combo box control (CSComboBox)

CSComboBox, a combo box control available through Progress Software Corporation, offers several search modes to locate data in the list box portion of the control. It also provides several ways to populate the list box, both by typing in the data at design time and by programmatically setting the data at runtime.

Spin button control (CSSpin)

CSSpin, a spin button control available through Progress Software Corporation, allows the user to enter or edit a numeric value, similar to the ABL slider widget. Where a slider changes value continually as you move it, a spin button changes value each time your press it or continually as you press and hold it. There are actually two buttons in the control for increase and decrease of value. The Value property of the control can return values from -32768 to 32767, depending on the settings of the Min and Max properties.

You can also use the value-changing events of the CSSpin control (SpinDown and SpinUp) to move back and forth through a database query with or without reference to the Value property. If necessary, you can programmatically reset the Value property to spin through more query rows than the integer range of the Value property might allow.

Timer control (PSTimer)

PSTimer, a control available through Progress Software Corporation, allows you to program tasks that execute at regular time intervals. You execute these tasks in an event procedure for the PSTimer Tick event. The Tick event fires at a programmable time interval that you can set using the Interval property.

19

Sockets

ABL (Advanced Business Language) provides direct access to TCP/IP sockets, both unsecured sockets and secured sockets using the Transport Layer Security (TLS). Sockets provide a means to implement interprocess communications with both local and remote processes. Using ABL sockets, you can communicate with non-ABL processes, as well as other ABL processes. Thus, you can implement socket-based applications completely in ABL that otherwise require the use of C modules accessible only through the HLC or shared library interfaces.

These topics describe how you can access TCP/IP sockets directly from ABL. It assumes that you are familiar with basic TCP/IP socket programming and TLS.

Note: TLS incurs heavy performance penalties, depending on the client, server, and network resources and load. For more information, see *Introduction to Security and Auditing*.

For details, see the following topics:

- ABL for programming sockets
- Overview of tasks to implement ABL sockets
- Implementing an ABL socket server
- Implementing an ABL socket client
- Read, writing, and managing sockets on clients and servers
- Implementing ABL socket security
- Examples using ABL sockets

ABL for programming sockets

The following table lists the ABL elements that are either valid only for working with sockets or have special application in socket programming. The remaining sections explain how to use these elements.

Table 65: ABL for programming sockets

ABL element	Description
BYTES-READ	An INTEGER attribute on the socket object handle that returns the number of bytes read during the last invocation of the socket READ( ) method. If the last READ( ) method call on the socket failed, this attribute returns 0.
BYTES-WRITTEN	An INTEGER attribute on the socket object handle that returns the number of bytes written during the last invocation of the socket WRITE( ) method. If the last WRITE( ) method call on the socket failed, this attribute returns 0.
CONNECT	An event received on a server socket object handle that indicates that a socket client is trying to connect. This event, if handled by an input-blocking or PROCESS EVENTS statement, executes any CONNECT event handler defined for the server socket object. You can use this event handler to obtain the handle to the socket object with which the client is communicating.
CONNECT (*connection-parameters*)	A method on the socket object handle that connects a socket handle to a specified TCP/IP port on a specified host.
CONNECTED( )	A method on the socket object handle that indicates if a socket handle is currently connected to a port.
CREATE SERVER-SOCKET *server-socket-handle*	A statement that creates a server socket object with all attributes set to their default values, and stores its handle in a HANDLE variable.
CREATE SOCKET *socket-handle*	A statement that creates a socket object with all attributes set to their default values and stores its handle in a HANDLE variable.
DEFINE INPUT PARAMETER *socket-handle*	A statement that defines the single INPUT parameter to the CONNECT event procedure (specified using the SET-CONNECT-PROCEDURE( ) method). This parameter returns the handle to the socket object created when a socket server receives a CONNECT event, and which the socket server uses to communicate with the corresponding socket client.
DELETE OBJECT *handle*	A statement that you can use to delete a handle, including socket and server socket object handles. To delete a connected socket object, you must first disconnect it using the DISCONNECT( ) method. To delete a server socket object enabled to listen for connections, you must first disable it using the DISABLE-CONNECTIONS( ) method.
DISABLE-CONNECTIONS( )	

ABL element	Description
	A method on the server socket object handle that indicates that new connections are no longer accepted on the server socket.
DISCONNECT( )	A method on the socket object handle that terminates the connection between the socket object and the port to which it is connected.
ENABLE-CONNECTIONS (*connection-parameters*)	A method on the server socket object handle that specifies the TCP/IP port that ABL uses to listen for new connections. Once called, ABL automatically listens for and accepts new connections on the specified port.ENABLE-CONNECTIONS( ) also lets you specify the length of the pending-connection queue.
FIRST-SERVER-SOCKET	A HANDLE attribute on the SESSION object handle that returns the handle to the first entry in the chain of server socket handles for the session. Note that you can have only one server socket object in the list enabled to listen for events at one time.
FIRST-SOCKET	A HANDLE attribute on the SESSION object handle that returns the handle to the first entry in the chain of socket handles for the session.
GET-BYTES-AVAILABLE( )	A method on the socket object handle that indicates the number of bytes available for reading from the socket.
GET-SOCKET-OPTION (*option-name*)	A method on the socket object handle that returns the specified TCP socket option. ABL supports the following options: • TCP-NODELAY • SO-LINGER • SO-KEEPALIVE • SO-REUSEADDR • SO-SNDBUF • SO-RCVBUF • SO-RCVTIMEO For more information on these options, see *ABL Reference* and your TCP documentation.
LAST-SERVER-SOCKET	A HANDLE attribute on the SESSION object handle that returns the handle to the last entry in the chain of server socket handles for the session. Note that you can have only one server socket object in the list enabled to listen for events at one time.

ABL element	Description
LAST-SOCKET	A HANDLE attribute on the SESSION object handle that returns the handle to the last entry in the chain of socket handles for the session.
LOCAL-HOST	A CHARACTER attribute on the socket object handle that returns the IP address on the local machine where the socket object is connected.
LOCAL-PORT	An INTEGER attribute on the socket object handle that returns the local port number of the socket object.
NEXT-SIBLING	A HANDLE attribute on the socket and server socket object handle that returns the next entry in the list of socket or server socket handles created for the current OpenEdge session.
MEMPTR	The ABL data type used by the socket handle READ( ) and WRITE( ) methods to read and write data on a socket. A MEMPTR expression defines a memory region whose size you must allocate in bytes. MEMPTR functions and statements allow you to read and write data between most other ABL data types and the specified memory region. For more information on the MEMPTR data type, see Introduction to External Program Interfaces on page 413.
PREV-SIBLING	A HANDLE attribute on the socket and server socket object handle that returns the previous entry in the list of socket or server socket handles created for the current OpenEdge session.
PROCESS EVENTS	A statement that you can use to handle any pending CONNECT or READ-RESPONSE events. You can also use any input-blocking statement, such as the WAIT-FOR statement.
READ(*memptr-expression* , *position* , *bytes-to-read* [, *mode*])	A method on the socket object handle that reads data from the specified socket. The method specifies the MEMPTR memory region and byte position within the region to store the data, a number of bytes to read (and store) from the socket, and a read mode. The read mode indicates if the exact specified number of bytes must be read or up to the specified number of bytes can be read.
READ-RESPONSE	An event received on a socket object handle indicating that data is waiting on the socket to be read. This event, if handled by an input-blocking or PROCESS EVENTS statement, executes any READ-RESPONSE event procedure or method defined for the socket object. You can use this event procedure or method if you want to read data from the socket in an event-driven manner.[12]
REMOTE-HOST	A CHARACTER attribute on the socket object handle that returns the IP address of the remote machine with which a connected socket object is communicating.

ABL element	Description
REMOTE-PORT	An INTEGER attribute on the socket object handle that returns the number of the port on the remote machine with which a connected socket object is communicating.
SELF	A system handle that returns the handle of the object on which an event is handled in the context of an event procedure. In a CONNECT event procedure, this is the handle to the server socket that is responding to a CONNECT event. In a READ-RESPONSE event procedure, this is the handle to the socket that is responding to a READ-RESPONSE event.
SENSITIVE	A LOGICAL attribute on the socket and server socket object handle that indicates whether the object can receive events. Set to TRUE (receive events) by default.
Server socket object handle	A handle to a server socket object. This object allows you to listen for and accept TCP/IP connections on a given port.
SET-CONNECT-PROCEDURE(*event-internal-procedure* [, *procedure-context*])	A method on the server socket object handle that specifies the name of an internal procedure (*CONNECT event procedure*) to invoke when a CONNECT event occurs.
SET-CALLBACK ("READ-RESPONSE", routine-name [, routine-context])	A method on the socket object handle that specifies the name of the callback to invoke when a READ-RESPONSE event occurs.
SET-READ-RESPONSE -PROCEDURE(*event-internal-procedure* [, *procedure-context*])	A method on the socket object handle that specifies the name of an internal procedure (*READ-RESPONSE event procedure*) to invoke when a READ-RESPONSE event occurs.

[12] A socket handle receives a READ-RESPONSE event under the same conditions that cause the TCP/IP select function to indicate that a socket is ready to receive results. However, in ABL, you must read data on the socket to continue to receive the event. For more information, see Data detection using the event-driven model on page 590.

ABL element	Description
Socket object handle	A handle to a socket object. This object allows you to read or write data on a TCP/IP socket and to perform other TCP/IP socket actions.
`SET-SOCKET-OPTION ( name, arguments )`	A method on the socket object handle that sets the specified TCP socket option. ABL supports the following options: • `TCP-NODELAY` • `SO-LINGER` • `SO-KEEPALIVE` • `SO-REUSEADDR` • `SO-SNDBUF` • `SO-RCVBUF` • `SO-RCVTIMEO` For more information on these options, see *ABL Reference* and your TCP documentation.
`SSL-SERVER-NAME`	A `CHARACTER` attribute on the socket object handle that returns the TLS server's X.500 Subject name field after any TLS session has been established.
`TYPE`	A `CHARACTER` attribute on the socket and server socket object handle that returns the handle type, which is `SERVER-SOCKET` for a server socket handle and `SOCKET` for a socket handle.
`WAIT-FOR ...`	A statement that you can use to handle any pending `CONNECT` or `READ-RESPONSE` events. You can also use `PROCESS EVENTS` or any other input-blocking statement, such as the `PROMPT-FOR` statement, to handle the events. When a `CONNECT` or `READ-RESPONSE` event occurs in the context of these statements, any `CONNECT` or `READ-RESPONSE` event procedure specified for the corresponding handle is executed.
`WRITE( memptr-expression , position , bytes-to-write )`	A method on the socket object handle that writes data to the specified socket. The method specifies the `MEMPTR` memory region and byte position within the region from which to write the data, as well as the number of bytes to write from the region.

Overview of tasks to implement ABL sockets

Socket programming can be typically divided into three parts:

- Server programming tasks
- Client programming tasks

- Tasks common to both clients and servers

This section summarizes the tasks for programming sockets in ABL using the same framework. The sections that follow describe each of these tasks in more detail.

The following table lists socket programming tasks by application function, indicates whether each task is required for implementing socket communications, and provides a reference to a following section for more information on the task.

Table 66: Summary of socket programming tasks

Application function	Tasks
Socket server	• Create a server socket object and enable it to listen for connections (`CONNECT` events) on a specified port (required). See Implementing an ABL socket server on page 588. • Define a `CONNECT` event procedure or method to respond to `CONNECT` events and return the socket object handle created for each connection (required). See Listening and responding to connection requests on page 588. • Make the server socket temporarily insensitive to `CONNECT` events. See Managing the server socket on page 589. • Disable the server socket from listening for new socket connections. See Managing the server socket on page 589. • Delete the server socket object. See Managing the server socket on page 589.
Socket client	• Create a socket object to read and write data, and connect it to a port on a socket server (required). See Implementing an ABL socket client on page 589.

Application function	Tasks
Client or server	• Define and initialize a MEMPTR variable (required to read or write data on a socket). See Defining and initializing MEMPTR variables on page 590. • Test whether data is available to read on a socket. See Data detection using the procedural model on page 590. • Define a READ-RESPONSE event procedure or method to respond to data coming in on a socket connection. See Data detection using the event-driven model on page 590. • Read data on a socket and test for the number of bytes read for each read operation. See Reading data on a socket on page 591. • Write data on a socket and test for the number of bytes written for each write operation. see Writing data on a socket on page 593. • Marshall data into a MEMPTR buffer for writing on a socket and unmarshall data from a MEMPTR buffer after reading on a socket. see Marshalling and unmarshalling data for socket I/O on page 594. • Test whether a socket object is connected. see Managing sockets and their connections on page 594. • Make a socket temporarily insensitive to READ-RESPONSE events. see Managing sockets and their connections on page 594. • Set and get socket options. see Managing sockets and their connections on page 594. • Obtain the local and remote host IP addresses and port numbers involved in a socket connection. see Managing sockets and their connections on page 594. • Disconnect a socket object from its remote host and port. See Managing sockets and their connections on page 594. • Delete socket objects. see Managing sockets and their connections on page 594.
Socket security	• Configure socket servers and clients for TLS. See Implementing ABL socket security on page 595. • Enable TLS server connections. See Enabling TLS server connections on page 596. • Connect an TLS client to a TLS server. See Connecting an TLS client to an TLS server on page 597. • Invoke socket operations. See Invoking socket operations for TLS sessions on page 598. **Note:** TLS incurs heavy performance penalties, depending on the client, server, and network resources and load.

Implementing an ABL socket server

This section describes how to implement an ABL socket server.

To implement a socket server:

1. Create a server socket object using the CREATE SERVER-SOCKET statement.
2. Enable the server socket to listen for connections using the ENABLE-CONNECTIONS( ) method. This assigns the socket to a specified TCP/IP port on which ABL listens and accepts connection requests.

   Note: ENABLE-CONNECTIONS( ) also lets you set the length of the pending-connection queue. For more information, see the reference entry for the ENABLE-CONNECTION( ) method in *ABL Reference*.

3. Specify a CONNECT event procedure or method using the SET-CALLBACK( ) method.
4. Wait for CONNECT events using an input-blocking statement or poll for CONNECT events using the PROCESS EVENTS statement.

   Once you have a connection established from a client, you can read data from and write data to the client on the connection. You can also manage the server socket that you have enabled for listening.

   The sections that follow describe these steps in more detail.

Listening and responding to connection requests

After you enable a server socket for listening using the ENABLE-CONNECTIONS( ) method, AVM listens on the specified port for client connections. When a socket client sends a connection request to this port, AVM automatically accepts the connection.

When a connection is made on a server socket the AVM creates a new socket for reading and writing to the socket. Also a CONNECT event is fired and the AVM calls any event procedure or method that you have specified using the SET-CALLBACK( ) function. The handle to the new read/write socket is passed into the callback handler as an input parameter. In the callback handler for the CONNECT event, the SELF system handle points to the server socket.

Note: In the callback handler for the READ-RESPONSE event, the SELF system handle points to the client socket.

To listen for connections on a server socket using the CONNECT event procedure:

1. Define an internal procedure that takes one INPUT parameter of type HANDLE to serve as an event procedure. You can define this procedure in any procedure context that is active while listening for connections.

2. Specify the procedure you defined in Step 1 as a CONNECT event procedure by invoking the SET-CONNECT-PROCEDURE( ) method on the server socket handle that you have enabled for listening.

3. Include input-blocking statements (such as WAIT-FOR) or PROCESS EVENTS statements in your code to handle events. When any CONNECT event is received in the context of one of these statements, the event procedure specified in Step 2 executes.

To listen for connection on a server socket using the CONNECT event method:

1. Define a public method that takes one `INPUT` parameter of type `HANDLE` to serve as an event method. You can define this public method in any class that is active while the server is listening for connections.
2. Specify the method you defined in Step 1 as a `CONNECT` event method by invoking the `SET-CALLBACK( )` method on the server socket object that you have enabled for listening.
3. Include input-blocking statements (such as `WAIT-FOR`) or `PROCESS EVENTS` statements in your code to handle events. When a `CONNECT` event is received in the context of one of these statements, the event method specified in Step 2 executes.

Once you have the handle to a connected socket object, you can read and write data on the socket, and otherwise manage the socket for the connection. For more information, see Read, writing, and managing sockets on clients and servers on page 589.

Managing the server socket

After you create and enable a server socket to listen for and accept connections, you can perform the following management functions on the server socket:

- **Control event sensitivity** — At any time, you can make the server socket stop receiving `CONNECT` events by setting its `SENSITIVE` attribute to `FALSE`. You can, at any time, return it to listen for `CONNECT` events by setting `SENSITIVE` to `TRUE`. Thus, when `SENSITIVE` is `FALSE`, ABL still listens on the specified port, but temporarily stops accepting connections and posting `CONNECT` events.
- **Disable listening for connections** — You can permanently stop the server socket from accepting connections by invoking its `DISABLE-CONNECTIONS( )` method. This stops the server socket from listening on the current port for new connections. However, all currently-connected sockets remain connected. If a client attempts to connect on the port, it receives an error.
- **Delete the server socket object** — You can delete a server socket object using the `DELETE OBJECT` statement. However, you must disable the server socket from listening for and accepting connections before you can delete it.

Implementing an ABL socket client

This section describes how to implement an ABL socket client.

To implement a socket client:

1. Create a socket object using the `CREATE SOCKET` statement.
2. Connect the socket object to the host and listening port of a socket server using the `CONNECT( )` method on the socket handle.
3. Read and write data on the socket and otherwise manage the socket.

   The tasks required to read and write data on a socket and to manage the socket are identical for both socket clients and socket servers. For more information, see Read, writing, and managing sockets on clients and servers on page 589.

Read, writing, and managing sockets on clients and servers

To read and write data on a socket object, you can use one of the following models, or a combination of both:

- **Procedural model** — Where you poll for and read data wherever necessary using the `READ( )` method on the socket handle and write data wherever necessary using the `WRITE( )` method on the socket handle.

- **Event-driven model** — Where you wait for a READ-RESPONSE event on the socket handle. You then read and write whatever data you need on the socket handle within an event procedure or method that executes in response to this event.

Regardless of the model you use, you must first define and initialize a MEMPTR variable to hold the data you want to read or write.

Defining and initializing MEMPTR variables

As with any ABL variable, you define a MEMPTR variable using the DEFINE VARIABLE statement or the VAR statement. Unlike most other data types, MEMPTR variables also have two other features that you must initialize to use them for socket communications:

- The size of the memory region reserved for the variable
- The byte order that ABL is to use to interpret certain data types that you can store in the variable

Before you can read or write to a MEMPTR variable, you must set the size of memory reserved for it, in bytes, using the ABL SET-SIZE statement. Depending on your data and application, you might also have to specify the byte order for reading and writing.

Generally, when writing and reading data on a socket, the client and server must agree on the byte order so ABL can consistently interpret, on both ends, the order of bytes that comprise each data type stored in the MEMPTR region. Thus, the MEMPTR data type supports the SET-BYTE-ORDER statement and the GET-BYTE-ORDER function to set and get the byte order for a MEMPTR variable. For more information, see Introduction to External Program Interfaces on page 413.

Detecting data on a socket

You can detect data on a socket using either the procedural or event-driven models.

Data detection using the procedural model

To detect data procedurally, you can simply read whatever data is available on the socket object using the READ( ) method. Using this model, you do not use the READ-RESPONSE event, and ABL does not automatically notify your application when data is available to read on the socket. You can also check:

- To see if the socket is connected using the CONNECTED( ) method
- To see how many bytes are available to read, using the GET-BYTES-AVAILABLE( ) method

How you use these methods depends on your application requirements and the options that you choose on the READ( ) method. For more information on READ( ) method options, see Reading data on a socket on page 591.

Data detection using the event-driven model

To detect data using events, you set up an event handler for READ-RESPONSE events posted on a connected socket object. There are two options for implementing the event handler, depending on whether the event handler code is in a method or a procedure. The first option is preferable and can be used for both methods and procedures. The second option is strictly for procedures.

Option 1: Use SET-CALLBACK( )

1. Define a PUBLIC method or a non-PRIVATE internal procedure that takes no arguments to contain your event handler code. If you are using a method for a callback, the method must satisfy the following requirements:

- The method signature must not have any parameters. Overloaded methods are allowed, but only the method with no parameters is invoked on the READ-RESPONSE event.
- The method return type should be VOID. Any value returned from the method is ignored.

You can define this method or procedure in any context that is active during the connection. When this method or procedure executes in response to a READ-RESPONSE event, the SELF system handle returns the handle of the connected socket.

2. Specify the method or procedure you defined in Step 1 as a READ-RESPONSE event callback by invoking the SET-CALLBACK( ) method on the socket.
3. Include input-blocking statements (such as WAIT-FOR) or PROCESS EVENTS statements in your code to initiate the handling of events. When any READ-RESPONSE event is received in the context of one of these statements, the event method or procedure specified in Step 2 executes.

Option 2: Use SET-READ-RESPONSE-PROCEDURE( )

1. Define an internal procedure that takes no arguments to serve as an event procedure. You can define this procedure in any procedure context that is active during the connection. When this procedure executes in response to a READ-RESPONSE event, the SELF system handle returns the handle of the connected socket.
2. Specify the procedure you defined in Step 1 as a READ-RESPONSE event procedure by invoking the SET-READ-RESPONSE-PROCEDURE( ) method on the socket.
3. Include input-blocking statements (such as WAIT-FOR) or PROCESS EVENTS statements in your code to initiate the handling of events. When any READ-RESPONSE event is received in the context of one of these statements, the event procedure specified in Step 2 executes.

In the event-driven model, ABL can post a READ-RESPONSE event on a connected socket object for two reasons:

- **Data has arrived on the socket** — You can then read this data during execution of the event procedure (or any time while the socket remains connected) using the READ( ) method. You do not have to read all the bytes available on the socket. If any data remains after reading on the socket, ABL immediately posts another READ-RESPONSE event on the socket object for the available data.

  Note: After ABL posts the first READ-RESPONSE event for a new socket, ABL does not post another READ-RESPONSE event for the socket until you call the READ( ) method on the socket object. Thus, if you do not read data on the socket in the event procedure, you must make sure to do so elsewhere in your application if you want the application to respond to any additional events for the socket.

- **The socket is disconnected** — During execution of an event procedure called for a socket disconnection:
  - Calling the READ( ) method on the socket object returns an error
  - The value returned by a GET-BYTES-AVAILABLE( ) method invoked on the socket object is zero (0)
  - The value returned by a CONNECTED( ) method invoked on the socket object is FALSE

Reading data on a socket

To read data from a connected TCP/IP socket, you invoke the READ( ) method on the corresponding socket object (SELF:READ( ) within the corresponding READ-RESPONSE event procedure or method). You can invoke this method on a connected socket at any time to read data. However, the method blocks depending on the amount of data available on the socket, the reading mode that you use, and the timeout value (set by the SO-RCVTIMEO option of the SET-SOCKET-OPTION( ) method).

This is the syntax of the READ( ) method:

Syntax

```
READ(
      memptr-expression
    , position
    , bytes-to-read
      [ , mode ]
    )
```

The READ( ) method transfers data from the socket to the specified MEMPTR variable, *memptr-expression*, at the byte position in the MEMPTR region specified by the INTEGER expression, *position*. Exactly how much data the READ( ) method blocks to read depends on:

- The number of bytes specified by the INTEGER*bytes-to-read*
- The reading mode specified by the INTEGER*mode*
- The timeout value, set by the SO-RCVTIMEO option of the SET-SOCKET-OPTION( ) method

Specifying the read mode

You can specify *mode* using a compiler constant as shown in the following table.

Table 67: Socket reading modes

Compiler constant	Value	Description
READ-AVAILABLE	1	The READ( ) method blocks until at least one (1) byte is available on the socket and reads the number of bytes that are available up to a maximum of *bytes-to-read* bytes.
READ-EXACT-NUM	2	(Default) The READ( ) method blocks until *bytes-to-read* bytes are available to read from the socket.

Thus, you can have the READ( ) method block until exactly the specified number of bytes are read (the default), or until all available bytes are read up to a maximum number allowed.

The appropriate reading mode to use depends on your application requirements. Note, however, that if you specify READ-EXACT-NUM, the READ( ) method blocks until it reads the specified number of bytes (no matter how long it takes) or until the socket is disconnected (whatever happens first).

Specifying the timeout length

Besides setting the read mode, you can also set the amount of time READ( ) waits before timing out. To do so, use the SO-RCVTIMEO option of the SET-SOCKET-OPTION( ) method. If you do not set a timeout value, the default is for READ( ) to wait forever.

READ( )'s timeout behavior is affected by the interaction of the read mode and the timeout value, as the following table illustrates.

Table 68: Effect of read mode and timeout value on READ()'stimeout behavior

Read mode	With a timeout value	Without a timeout value
READ-AVAILABLE	Assuming there are no connection failures, READ( ) blocks until one of the following occurs: • The timeout expires • There is at least one byte available to read on the socket	Assuming there are no connection failures, READ( ) blocks until there is at least one byte available to read on the socket
READ-EXACT-NUM	Assuming there are no connection failures: • If there is any data on the socket, READ( ) blocks until there are *bytes-to-read* bytes available to read on the socket • Else, READ( ) blocks until the timeout expires	Assuming there are no connection failures, READ( ) blocks until there are *bytes-to-read* bytes available to read on the socket

Verifying the number of bytes actually read

You can verify the number of bytes actually read by the READ( ) method. The number of bytes read by the last READ( ) method invoked on a socket object is equal to the value of the BYTES-READ attribute invoked on the same socket object.

Writing data on a socket

You can write data on a connected TCP/IP socket at any time using the WRITE( ) method on the corresponding socket object (SELF:WRITE( ) within the corresponding READ-RESPONSE event procedure or method).

The is the syntax of the WRITE( ) method:

Syntax

```
WRITE(
      memptr-expression
    , position
    , bytes-to-write
    )
```

The WRITE( ) method transfers the number of bytes specified by the INTEGER expression, *bytes-to-write*, from the specified MEMPTR variable, *memptr-expression*, to the socket starting from the byte position within the MEMPTR region specified by the INTEGER expression, *position*.

You can verify the number of bytes actually written by the WRITE( ) method. The number of bytes written by the last WRITE( ) method invoked on a socket object is equal to the value of the BYTES-WRITTEN attribute invoked on the same socket object.

Marshalling and unmarshalling data for socket I/O

After reading data from a socket into a MEMPTR variable, any ABL procedure that needs to interpret the data must unmarshall the data from the MEMPTR memory region. Unmarshalling the data converts it from bytes in memory to the ABL data types that the ABL procedure can use, as determined by your application and the byte order of the MEMPTR data.

Similarly, before writing data to a socket from a MEMPTR variable, an ABL procedure must marshall data into the MEMPTR memory region. Marshalling the data converts it from the ABL data types the procedure understands to bytes in memory that can be written to the socket. Again, how you organize the bytes in the MEMPTR memory region depends on your application and the application with which you are communicating.

ABL supports several statements and functions for marshalling and unmarshalling data in different forms. All such statements and functions that interact directly with MEMPTR variables convert between bytes and ABL data types according to the MEMPTR byte order that you specify. Before using a MEMPTR variable to read or write socket data, both the socket client and socket server must set their respective MEMPTR variables to an identical byte order. When marshalling and unmarshalling the data, you must make certain to access MEMPTR data in conformance with the agreed and specified MEMPTR byte order.

For more information on byte order and the statements and functions for marshalling and unmarshalling MEMPTR data, see Introduction to External Program Interfaces on page 413.

Managing sockets and their connections

To manage a socket and its connection, ABL allows you to:

- **Test if the socket is connected** — If the CONNECTED( ) socket method is TRUE, the socket is connected.
- **Control event sensitivity** — At any time, you can make the socket stop receiving READ-RESPONSE events by setting its SENSITIVE attribute to FALSE. Also at any time, you can return it to receiving READ-RESPONSE events by setting SENSITIVE to TRUE.
- **Set socket options** — ABL supports the following options:
  - TCP-NODELAY
  - SO-LINGER
  - SO-KEEPALIVE
  - SO-REUSEADDR
  - SO-SNDBUF
  - SO-RCVBUF
  - SO-RCVTIMEO

  To set and get these options, use the SET-SOCKET-OPTION( ) and GET-SOCKET-OPTION( ) methods.
- **Obtain host and port values** — You can obtain the remote host IP address and port number involved in the connection from the values of the REMOTE-HOST and REMOTE-PORT socket attributes. Similarly, you can obtain the local host and port from the LOCAL-HOST and LOCAL-PORT socket attributes.
- **Disconnect a socket** — You can close the socket and remove the association between the socket object and its remote port by invoking the DISCONNECT( ) socket method. ABL automatically closes the local socket when ABL detects that the corresponding remote socket in a connection is closed.
- **Delete the socket object** — You can delete a socket object using the DELETE OBJECT statement. However, you must disconnect the socket before you can delete it.

Caution: Note that you should also delete the SELF handle you used to read and write to the socket to avoid a memory leak.

Implementing ABL socket security

You can use the Transport Layer Security (TLS) to provide a security infrastructure that protects communications between a socket client and server. TLS provides data privacy over network connections and authentication between clients and servers on those connections using elements of Public Key Infrastructure (PKI). These elements include private and public keys that the clients and servers use to authenticate each other and to set up data encryption and decryption services between the initiator of the communications (TLS client) and the responder (TLS server). The server is identified by the private key that it stores and the client is identified as a valid TLS client for that server by the public key that it stores and provides to the server. TLS clients gain access to public keys using digital (public key) certificates provided by a trusted certificate authority (CA) that also provides the private key confidentially to the TLS server.

Like socket communications in general, TLS is both application and transport independent. This section describes how to implement and manage TLS for ABL applications communicating over TCP/IP sockets.

For more information on TLS and how it uses private and public keys and public key certificates to handle security tasks in these contexts, see *Introduction to Security and Auditing*.

Note: TLS incurs heavy performance penalties, depending on the client, server, and network resources and load.

To set up and connect ABL socket servers and clients with TLS:

1. On your ABL socket server, create and configure the private key and certificate store required to identify your socket server as an TLS server.
2. On your ABL socket client, create and configure the public key certificate store required to access your TLS server as an TLS client.
3. Startup and ensure that your ABL socket server has enabled connections for TLS.
4. Startup and connect your ABL client as an TLS client to your TLS server.
5. Invoke socket operations in TLS sessions.

The sections that follow describe these tasks.

Configuring TLS servers and clients

OpenEdge provides utilities to create and manage key and certificate stores that enable your OpenEdge servers and clients to use the full capabilities of TLS. For all OpenEdge clients and servers, OpenEdge also installs built-in key and certificate stores that provide a simple, default TLS implementation for applications that require a minimal level of TLS support. TLS servers and clients have different TLS configuration requirements. So, if your TLS server is also an TLS client of some other TLS server, you must configure your application for both.

Note: Be very sure you need TLS before using this option. TLS incurs more or less heavy performance penalties, depending on resources and load.

Configuring TLS servers

For an TLS server, you must have installed a private key and digital (public key) certificate that uniquely identifies your ABL socket server as an TLS server and allows all communications to be encrypted between it and any TLS client. You can use the default key and certificate store provided by OpenEdge without any additional work. This provides default encryption services between all OpenEdge clients and servers and there by eliminates the need for client-server authentication to complete TLS connections.

However, to create a complete TLS implementation that supports all the features of TLS, you must obtain a unique private key and server digital certificate from an industry-recognized certificate authority (a CA such as Symantec) or create them yourself, as your own private CA, using server certificate administration software that you obtain on your own. Once you have the required private key and digital certificate, you can install it confidentially on your server system using the `pkiutil` command-line tool provided by OpenEdge. At this point your ABL socket server is ready to enable TLS connections.

For more information on OpenEdge TLS support, CA's, keys, digital certificates, and using `pkiutil`, see *Introduction to Security and Auditing*.

Configuring TLS clients

For an TLS client, you must have installed a public key certificate that allows the client to authenticate and encrypt communications with a specific TLS server that it connects to. You can obtain the required public key certificate for a given TLS server from the CA that issued the server's private key. Once you have the public key certificate, you can install it on your client system using the `certutil` command-line tool provided by OpenEdge. At this point your ABL socket client is ready to make an TLS connection to an TLS server.

For more information on OpenEdge TLS support, public key certificates, and using `certutil`, see *Introduction to Security and Auditing*.

Enabling TLS server connections

When you enable TLS connections on an ABL server socket object, the socket accepts connections only from TLS clients that can authenticate to your ABL application as an TLS server.

Note: Be very sure you need TLS before using this option. TLS incurs more or less heavy performance penalties, depending on resources and load.

To enable TLS connections on an ABL server socket object, include the TLS parameters shown in the following table as part of the connection parameters string that you pass to the server socket object `ENABLE-CONNECTIONS()` method. For additional information, see Implementing an ABL socket server on page 588.

Table 69: Server socket object TLS connection parameters

Connection parameter	Description
`-ssl`	Specifies the all connections to this server socket must use TLS.
`-keyalias` *`aliasname`*	Sets the alias name within the keystore of the private key and digital certificate entry to use to authenticate all connections to this server socket. If not specified, the server socket uses the `default_server` server certificate alias.
`-keyalaispasswd` *`encrypted-password`*	Sets the password to use for accessing the private key and digital certificate. You must specify a password when you specify the `-keyalias` option and the password must be encrypted. You must specify the password as an encrypted value that you can obtain using the `genpassword` utility located in the bin directory of your OpenEdge installation. If you use the `default_server` server certificate, it also has a default password that you do not need to specify.
`-nosessioncache`	If specified, caching for the TLS client session is disabled.
`-sessiontimeout` **[** *`seconds`* **]**	Specifies, in seconds, the length of time that an TLS client session is held in the session cache, during which an TLS client can resume its session. The default is 180 seconds.

For more information on OpenEdge TLS support, key and certificate stores, the `genpassword` utility, and enabling connections to TLS servers, see *Introduction to Security and Auditing*.

Connecting an TLS client to an TLS server

When you connect an ABL socket object to an TLS server socket, you must specify the connection as an TLS connection. Your connecting client must be able to authenticate itself to the TLS server, and you can optionally require the server to authenticate itself to your TLS client.

Note: Be very sure you need TLS before using this option. TLS incurs more or less heavy performance penalties, depending on resources and load.

To connect an ABL socket object to an TLS server socket, include the TLS parameters shown in the following table as part of the connection parameters string that you pass to the socket object `CONNECT( )` method. For additional information, see Implementing an ABL socket client on page 589.

Table 70: Client socket object TLS connection parameters

Connection parameter	Description
`-ssl`	Directs that the connection to the server socket use TLS tunneling.

Connection parameter	Description
-nosessionreuse	If specified, the connection does not reuse the TLS session ID when reconnecting to the same TLS server.
-nohostverify	If specified, turns off host verification for an TLS connection. Without this parameter specified, the client compares the host name by the -H parameter with the Common Name specified in the server digital certificate, and raises an error if they do not match. With this parameter specified, the client never raises the error.

For more information on OpenEdge TLS support and connecting an TLS client to an TLS server, see *Introduction to Security and Auditing*.

Invoking socket operations for TLS sessions

For working in a TLS session, OpenEdge provides an additional client-side feature and also presents a potential deadlock condition when the server, but not the client, is connected using TLS.

Identifying the TLS server in a client session

After connecting to an TLS server socket from a client socket object using TLS, the client can identify the server by reading the value of the SSL-SERVER-NAME attribute on the socket object handle. This CHARACTER attribute returns the value of the SSL server's X.500 Subject name field from the server certificate after any SSL session has been established for a given socket object.

Deadlocking when a non-TLS client socket connects to an TLS server socket

When you open a non-TLS connection from an ABL client to an ABL TLS server, you can initiate a situation that can result in an deadlock condition that is unbreakable from with in the application. This condition results from the following sequence of events:

1. The non-TLS ABL client connects a socket to a server socket on an ABL server using the CONNECT( ) method.
2. The client immediately follows the CONNECT( ) method by a READ( ) method on the socket object.
3. The ABL server enables TLS connections using the ENABLE-CONNECTIONS( ) method.
4. The server immediately follows the ENABLE-CONNECTIONS( ) method by a WRITE( ) method on the server socket object.

After Step 2, the non-TLS client begins waiting for the socket server to send it data. After Step 4, the socket server waits in a read mode (despite the WRITE( )) for the client to begin an TLS connection operation. This results in both ABL sessions waiting for data from the other. For both the ABL client and server, the application hangs because ABL does not provide a socket timeout feature.

There is no way to detect this situation once the deadlock has occurred. However, the basic strategy for handling it is to know ahead of time that the server is running an TLS session. You can use the following techniques to determine the state of the server before continuing from the client:

- On the ABL socket server, use the SET-CONNECT-PROCEDURE( ) method to handle a CONNECT event asynchronously and watch a manual timer to determine if the ABL socket server is hanging.
- On the client side, you can also use a small procedure to test the connection before continuing with running the application.

Examples using ABL sockets

The following sample procedures show a simple interaction between a socket server (`serversocket.p`) and a socket client (`socketclient.p`). In this case, the socket server exits after handling a single connection and the socket client exits after receiving one data transmission from the server. These are event-driven examples, where the socket server only writes and the socket client only reads on their respective sockets.

The following sample procedures show a simple interaction between a socket server (`serversocket.p`) and a socket client (`socketclient.p`). In this case, the socket server exits after handling a single connection and the socket client exits after receiving one data transmission from the server. These are event-driven examples, where the socket server only writes and the socket client only reads on their respective sockets. The methods for the callback events CONNECT and READ-RESPONSE are defined in the class definition (`SocketManager.cls`).

SocketManager.cls

```
CLASS SocketManager:

METHOD PUBLIC VOID OnReadResponse():

  VAR HANDLE readSocket.
  VAR MEMPTR clientMessage.
  VAR CHARACTER msgStr.

  readSocket = SELF.

  IF readSocket:get-bytes-available() = 0 THEN DO:
    RETURN.
  END.

  SET-SIZE(clientMessage) = readSocket:GET-BYTES-AVAILABLE().

  readSocket:READ(clientMessage, 1, GET-SIZE(clientMessage)).

  msgStr = GET-STRING(clientMessage, 1, GET-SIZE(clientMessage)).

  SET-SIZE(clientMessage) = 0.

  MESSAGE "Message from serverSocket: "  msgStr SKIP.

END METHOD.

METHOD PUBLIC VOID OnClientConnect ( INPUT vSocketHandle AS HANDLE):
  VAR CHARACTER vGt = "Server Response: got a client connection and this is the response
 message".
  VAR memptr vGB.

  // send message to client
  SET-SIZE(vGB) = LENGTH(vGt) + 1.
  PUT-STRING(vGB, 1) = vGt.
  vSocketHandle:write(vGB, 1, get-size(vGB)).
  SET-SIZE(vGB) = 0.
END METHOD.

END CLASS
```

serversocket.p

```
VAR HANDLE serverSocket.
VAR SocketManager callbackObj.

// create a socket
CREATE server-socket serverSocket.

//Create an object instance of the class(es) containing the event methods
callbackObj = NEW SocketManager().

// When someone connects to this socket, run the method "onClientConnect"
serverSocket:set-callback("CONNECT", "OnClientConnect", callbackObj).
serverSocket:enable-connections("-S 44120").

// keep socket going - duration is just a single request
WAIT-FOR CONNECT of serverSocket.

serverSocket:DISABLE-CONNECTIONS().

DELETE OBJECT serverSocket.
DELETE OBJECT callbackObj.
QUIT.
```

socketclient.p

```
VAR HANDLE clientsocket.
VAR SocketManager callbackObj.

//create a socket for the client connection
CREATE SOCKET clientsocket.

callbackObj = NEW SocketManager().
clientsocket:set-callback("READ-RESPONSE", "OnReadResponse", callbackObj).
clientsocket:connect("-H localhost -S 44120").

IF clientsocket:CONNECTED() THEN
    MESSAGE "Client successfully connected to the server socket" SKIP.
ELSE DO:
    MESSAGE "Client failed to connect to the server socket - fatal error, test aborted"
 SKIP.
  QUIT.
END.

WAIT-FOR READ-RESPONSE OF clientsocket.

clientsocket:DISCONNECT().
DELETE OBJECT clientsocket.
DELETE OBJECT callbackObj.
QUIT.
```

The following sample procedures show how you can transfer a database record between a socket server (`i-sktsv2.p`) and a socket client (`i-sktcl2.p`) using a raw transfer. It uses the `sports2000` database as the source (server) and a copy of the `sports2000` database as the target (client).

This is a simple example in which the server waits for connections indefinitely (until you press the **STOP** key), but always sends the same database record to the client for each `CONNECT` event. Note how the server uses the `RAW` transfer to first copy the record from the database, then to specify the size of the `MEMPTR` memory from which the record is written on the socket. The server stores the size of the record as the first piece of information sent to the client, followed by the record.

Note: The WAIT FOR statements in the following samples are based on simple scenarios. Real life applications you would probably return back to the existing WAIT-FOR that drives the entire application. In which case the event handler would have to be in a persistent mode and not as used in the samples below. For more information, see "WAIT FOR statement" in the *ABL Reference*.

---

i-sktsv2.p

```
/* Sends new customer record from the sports2000 database through a socket. */
DEFINE VARIABLE aOk           AS LOGICAL NO-UNDO.
DEFINE VARIABLE custNum       AS INTEGER NO-UNDO.
DEFINE VARIABLE hServerSocket AS HANDLE  NO-UNDO.
DEFINE VARIABLE len           AS INTEGER NO-UNDO.
DEFINE VARIABLE mBuffer       AS MEMPTR  NO-UNDO.
DEFINE VARIABLE r1            AS RAW     NO-UNDO.

MESSAGE "We are in the raw transfer socket server".

/* Create a new customer record with a unique custnum */
FIND LAST Customer NO-LOCK.
custNum = Customer.CustNum + 1.

CREATE Customer.
ASSIGN
  Customer.CustNum    = custNum
  Customer.Name       = "Jack Sprat"
  Customer.Address    = "222 Ferdinand St"
  Customer.City       = "Woburn"
  Customer.State      = "MA"
  Customer.PostalCode = "01801".

CREATE SERVER-SOCKET hServerSocket.
hServerSocket:SET-CALLBACK("CONNECT", "connProc", THIS-PROCEDURE).

/* Put customer record into a RAW variable and store in local DB. */
RAW-TRANSFER customer TO r1.
RELEASE customer.

/* Put the length of the record followed by a copy of the full record into a
   MEMPTR to send it through the socket. */
ASSIGN
  len                     = LENGTH(r1)
  SET-BYTE-ORDER(mBuffer) = LITTLE-ENDIAN
  SET-SIZE(mBuffer)       = len + 4
  PUT-LONG(mBuffer, 1)    = len
  PUT-BYTES(mBuffer, 5)   = r1.

aOk = hServerSocket:ENABLE-CONNECTIONS( "-S 3334").
MESSAGE "Enabled connections:" aOk.

DO ON STOP UNDO, LEAVE:
  WAIT-FOR CLOSE OF THIS-PROCEDURE.
END.

DELETE OBJECT hServerSocket.

PROCEDURE connProc:
  /* Connection procedure for server socket */
  DEFINE INPUT PARAMETER hSocket AS HANDLE.  /*Socket implicitly created*/
  IF VALID-HANDLE(hSocket) THEN DO: /* standard validation */
    MESSAGE "We are in the CONNECT event procedure, connProc".
    /* Send the size followed by the record that we put into the MEMPTR */
    hSocket:WRITE (mBuffer,1, GET-SIZE(mBuffer)).
    MESSAGE "Sent" hSocket:BYTES-WRITTEN "bytes containing customer record".
    hSocket:DISCONNECT().
    DELETE OBJECT hSocket.
  END.
  ELSE MESSAGE "Unable to obtain socket.".
END PROCEDURE.
```

In this example, the client (`i-sktcl2.p`) polls its socket procedurally until the data for the record is available to read. In this case, the client first waits for the size information, then waits for that number of bytes of Customer data. It also uses this size information to set the size of the `MEMPTR` region for reading the record off the socket. Finally, note that the client deletes the socket object and frees `MEMPTR` memory after it disconnects from the server.

i-sktcl2.p

```
/* Receives a new customer record through a socket and puts it into the
   sports2000 DB. */
DEFINE VARIABLE hSocket AS HANDLE  NO-UNDO.
DEFINE VARIABLE mBuffer AS MEMPTR  NO-UNDO.
DEFINE VARIABLE r1      AS RAW     NO-UNDO.
DEFINE VARIABLE recLen  AS INTEGER NO-UNDO.

MESSAGE "We are in the raw transfer socket client".

CREATE SOCKET hSocket.
hSocket:CONNECT ("-H localhost -S 3334").
IF hSocket:CONNECTED() THEN
  MESSAGE "Connected OK".
ELSE DO:
  MESSAGE "Could not connect".
  RETURN.
END.
/* Do a blocking READ until we get the count of how long the record is to
   read. */
SET-SIZE(mBuffer)       = 4.
SET-BYTE-ORDER(mBuffer) = LITTLE-ENDIAN.
hSocket:READ(mBuffer, 1, 4).
recLen = GET-LONG(mBuffer, 1).

/* READ again to get the full record. */
SET-SIZE(mBuffer) = 0.
SET-SIZE(mBuffer) = recLen.
hSocket:READ(mBuffer, 1, recLen).
MESSAGE hSocket:BYTES-READ "bytes read for customer record".

/* Copy the record to a RAW variable so we can put it into the DB */
r1 = mBuffer.
RAW-TRANSFER r1 TO Customer.

MESSAGE "Customer is:" Customer.CustNum Customer.Name Customer.Address
  Customer.City Customer.State Customer.PostalCode.
RELEASE Customer.

hSocket:DISCONNECT().
DELETE OBJECT hSocket.
SET-SIZE(mBuffer) = 0.
```

The next example also involves a client and server. The server, `i-sktsv3.p`, demonstrates how to set `qsize`, the length of the pending-connection queue, while enabling the server-socket for connections.

i-sktsv3.p

```
DEFINE VARIABLE m-string AS MEMPTR  NO-UNDO.
DEFINE VARIABLE ok       AS LOGICAL NO-UNDO.
DEFINE VARIABLE sserver  AS HANDLE  NO-UNDO.

CREATE SERVER-SOCKET sserver.
ok = sserver:SET-CALLBACK("CONNECT", "connProc").
MESSAGE "set connection procedure" ok.

/* Enable for connections and set the qsize */
ok = sserver:ENABLE-CONNECTIONS( "-S 3000 -qsize 5").
MESSAGE "enable connections" OK.

WAIT-FOR CONNECT OF sserver.

PROCEDURE connProc.
  /* connection procedure for server socket */
  DEFINE INPUT PARAMETER hsocket AS HANDLE.  /* implicitly created */

  IF VALID-HANDLE(hSocket) THEN DO: /* standard validation */
    MESSAGE "Inside connProc".
    SET-SIZE(m-string)     = 64.
    PUT-STRING(m-string,1) = "SERVER - hello".
    ok = hsocket:WRITE (m-string,1,26).
    hSocket:DISCONNECT().
    DELETE OBJECT hSocket.
  END.
  ELSE MESSAGE "Unable to obtain socket.".
END.
```

The client, `i-sktcl3.p`, shows how to set and retrieve socket options.

i-sktcl3.p

```
DEFINE VARIABLE hsocket  AS HANDLE    NO-UNDO.
DEFINE VARIABLE ok       AS LOGICAL   NO-UNDO.
DEFINE VARIABLE ret      AS CHARACTER NO-UNDO.
DEFINE VARIABLE m-string AS MEMPTR    NO-UNDO.
DEFINE VARIABLE c        AS CHARACTER NO-UNDO.

CREATE SOCKET hsocket.

hsocket:CONNECT ("-H localhost -S 3000 ").
IF hsocket:CONNECTED() THEN
  MESSAGE "Connected OK".
ELSE DO:
  MESSAGE "Could not connect".
  RETURN.
END.
/* connect succeeded */

OK  = hsocket:SET-SOCKET-OPTION("TCP-NODELAY", "TRUE").
ret = hsocket:GET-SOCKET-OPTION("TCP-NODELAY").
MESSAGE "TCP-NODELAY = " ret.

OK  = hsocket:SET-SOCKET-OPTION("SO-LINGER", "TRUE, 55").
ret = hsocket:GET-SOCKET-OPTION("SO-LINGER").
MESSAGE "SO-LINGER = " ret.

OK  = hsocket:SET-SOCKET-OPTION("SO-KEEPALIVE", "TRUE").
ret = hsocket:GET-SOCKET-OPTION("SO-KEEPALIVE").
MESSAGE "SO-KEEPALIVE = " ret.

OK  = hsocket:SET-SOCKET-OPTION("SO-REUSEADDR", "TRUE").
ret = hsocket:GET-SOCKET-OPTION("SO-REUSEADDR").
MESSAGE "SO-REUSEADDR = " ret.

OK  = hsocket:SET-SOCKET-OPTION("SO-SNDBUF", "40").
ret = hsocket:GET-SOCKET-OPTION("SO-SNDBUF").
MESSAGE "SO-SNDBUF = " ret.

OK  = hsocket:SET-SOCKET-OPTION("SO-RCVBUF", "5000").
ret = hsocket:GET-SOCKET-OPTION("SO-RCVBUF").
MESSAGE "SO-RCVBUF = " ret.

OK  = hsocket:SET-SOCKET-OPTION("SO-RCVTIMEO", "60").
ret = hsocket:GET-SOCKET-OPTION("SO-RCVTIMEO").
MESSAGE "SO-RCVTIMEO = " ret.

WAIT-FOR READ-RESPONSE OF hsocket.
SET-SIZE(m-string) = 64.
ok = hsocket:READ(m-string,1,26).
c  = GET-STRING(m-string,1).
MESSAGE "the string = " c.
hsocket:DISCONNECT().
DELETE OBJECT hsocket.
```

Socket example using a method as a callback

The following code examples show an interaction between a socket server (`serversocket.p`) and a socket client (`socketclient.p`). This is an event-driven example, where the event handler code is contained in a method called `OnReadResponse` within a class file (`callbackclass.cls`).

serversocket.p

```
define variable serverSocket as handle no-undo.

// create a socket
create server-socket serverSocket.

// When someone connects to this socket, run the procedure "onClientConnect"
serverSocket:set-connect-procedure("onClientConnect").
serverSocket:enable-connections("-S 44120").

// keep socket going - duration is just a single request
wait-for connect of serverSocket.

serverSocket:disable-connections().
delete object serverSocket.
quit.

procedure onClientConnect:
    define input parameter vSocketHandle as handle no-undo.
    define variable vGt as character no-undo init "Server Response: got a client
connection and this is the response message".
    define variable vGB as memptr no-undo.

    // send message to client
    set-size(vGB) = length(vGt) + 1.
    put-string(vGB, 1) = vGt.
    vSocketHandle:write(vGB, 1, get-size(vGB)).
    SET-SIZE(vGB) = 0.

end procedure.
```

socketclient.p

```
define variable clientsocket as handle no-undo.

create socket clientsocket.

define variable callbackClass as callbackclass no-undo.

callbackClass = new callbackclass().
clientsocket:set-callback("READ-RESPONSE", "OnReadResponse", callbackClass).
clientsocket:connect("-H localhost -S 44120").

if clientsocket:connected() then
   message "Client successfully connected to the server socket" skip.
else
do:
   message "Client failed to connect to the server socket - fatal error, test aborted"
 skip.
   quit.
end.

wait-for read-response of clientsocket.

clientsocket:disconnect().
delete object clientsocket.
quit.
```

callbackclass.cls

```
class callbackclass:

   method public void OnReadResponse():
```

```
      define variable readSocket as handle no-undo.
      define variable clientMessage as memptr no-undo.
      define variable msgStr as character no-undo.

      readSocket = self.

      if readSocket:get-bytes-available() = 0 then do:
        return.
      end.

      set-size(clientMessage) = readSocket:get-bytes-available().

      readSocket:read(clientMessage, 1, get-size(clientMessage)).

      msgStr = get-string(clientMessage, 1, get-size(clientMessage)).

      set-size(clientMessage) = 0.

      message "Message from serverSocket: "  msgStr skip.

    end method.

end class.
```

A

COM Object Data Type Mapping

OpenEdge automatically converts data between COM data types and ABL (Advanced Business Language) data types for COM object methods and properties, as well as for ActiveX control event parameters. This appendix describes this conversion process and the general support for COM data types in ABL. To perform conversions, ABL uses:

- Options in ABL that indicate COM data types
- The Type Libraries installed with each Automation Server or ActiveX control
- The data structures that define the data for each COM object property and parameter

For more information on the syntax for COM object references and ABL support for Type Libraries, see Using COM Objects in ABL on page 523.

The data conversion support for ActiveX control event parameters is a subset of the general support for COM object data type mapping. For information on this support, see ActiveX Control Support on page 551.

For details, see the following topics:

- Data type conversion strategy
- Conversions from COM to ABL data types
- Conversions from ABL to COM data types
- Alternate COM data type names

Data type conversion strategy

OpenEdge uses different mechanisms to convert COM data types to ABL data types and to convert ABL data types to COM data types.

Converting COM to ABL data types

A COM object passes data to ABL in a Variant structure for COM object properties (get), method output parameters, method return values, and event input parameters. A *Variant* is a self-describing data type whose structure contains both the type of data and the data itself. ABL accesses the information in this structure to convert COM data types to ABL data types.

Converting ABL to COM data types

ABL uses ABL type specifiers and the object's Type Library to convert ABL data types to COM data types for COM object properties (set), method input parameters, and event output parameters. This is the procedure that ABL follows for these conversions:

1. If ABL includes type specifiers, ABL performs the conversion based on these specifiers.
2. If ABL does not include type specifiers, ABL checks if the COM object provides any type information through its Type Library interfaces. If available, ABL performs the conversion based on that type information.
3. If neither Step 1 nor Step 2 results in a conversion, ABL performs the default conversion.

For information on ABL type specifiers and how to use them, see Using COM Objects in ABL on page 523.

General conversion features and limitations

If you set a method input parameter, an event output parameter, or a property to the Unknown value (?), ABL converts this value to the COM Null value. Similarly, if a COM object returns a Null value in a method output parameter, an event input parameter, or a property, ABL converts this value to the Unknown value (?).

When the OpenEdge COM Object Viewer suggests the syntax for a property or method reference or the AppBuilder generates a template for an OCX event procedure, it indicates any method parameter or property with an undetermined data type using the place holder, *anytype*. You must replace *anytype* with a valid ABL data type in your code. For more information on the OpenEdge COM Object Viewer, see Using COM Objects in ABL on page 523. For more information on OCX event procedure templates, see ActiveX Control Support on page 551.

Conversions from COM to ABL data types

The following table lists the possible conversions from COM data types to ABL data types when accessing COM object properties, method return values, method output parameters, and event input parameters.

If the destination ABL data item has a different data type than the one listed in the following table, ABL tries to convert it.

Table 75: Default conversions from COM data types to ABL data types

COM data type	ABL data type
Boolean	LOGICAL
Signed Byte	INTEGER
Unsigned Byte	INTEGER
Signed Short (2-byte integer)	INTEGER
Unsigned Short (2-byte integer)	INTEGER
Signed Integer (4-byte integer)	INTEGER
Unsigned Integer (4-byte integer)	INT64
Signed 8-byte Integer	INT64
Unsigned 8-byte Integer	DECIMAL
Byte[] (Array)	RAW
Currency	DECIMAL
Date	DATETIME
Double	DECIMAL
Error	INTEGER
Float (Single)	DECIMAL
Object	COM-HANDLE
String	CHARACTER

Note:

The names for COM data types in the above table conform to a nomenclature used in the documentation for most COM objects. For matching alternatives to these names, seen in some documentation and COM object viewers, see Table 77: Alternative COM data type names on page 614.

ABL resolves any Pointer or Variant Pointer references to COM data types, then converts the value to the corresponding ABL data type.

Conversions from ABL to COM data types

The following table lists the possible conversions from ABL data types to COM data types when setting COM object properties, passing method input parameters, or returning event output parameters.

If there is no type information from the COM object and there is no explicit data type specifier in the ABL code (for example, AS SHORT) ABL performs a default conversion as shown in the following table. However, in addition to the listed conversion pairs, ABL can convert other ABL data types to COM data types when the target COM data type is known and the data is compatible. Typically, for example, an ABL INTEGER would be used for a COM parameter defined as Short. In this and other similar conversions, overflow may occur resulting in a run time error condition.

Table 76: Conversions from ABL to COM data types

ABL data type	COM data type
CHARACTER	String
COM-HANDLE	Object
DATE	Date
DATETIME	Date
DATETIME-TZ	Date
DECIMAL	Double
INTEGER	Signed Integer (4 bytes)
INT64	Signed 8-byte Integer
LOGICAL	Boolean
LONGCHAR	String
RAW	Array of bytes
anytype	Variant

Note:

Non-obvious data type conversions may be possible, but are not recommended. For example, a character string with the value "123" can be converted to a Currency COM data type, even though the above table does not show a default CHARACTER to Currency conversion pair.

The names for COM data types in the above table conform to a nomenclature used in the documentation for most COM objects. For matching alternatives to these names, seen in some documentation and COM object viewers, see Table 77: Alternative COM data type names on page 614.

ABL to COM data type features and limitations

When performing ABL to COM data type conversions, ABL does do array and automatic pointer conversions, but does not do conversions for certain data ABL types.

Array conversions

ABL supports conversions between COM Array data types and ABL arrays (data types with extents like AS INTEGER EXTENT 5.) The conversion for arrays follows the conversions listed in Table 76: Conversions from ABL to COM data types on page 612.

In addition, COM arrays, including those returned from methods and properties of ActiveX or Automation objects, can be used in ABL anywhere an ABL EXTENT object is permitted. For example:

```
dateArray = ComObj:GetHolidays().

comObj:Holidays = dateArray.

ComObj:SetDates(ComObj2:GetHolidays()).

RUN SetDates(ComObj2:GetHolidays()).
```

Note: Attempting to assign a COM array of Variants, where the Variants are not a homogeneous type results in a runtime error.

The COM object viewer will show the return value of a method or property that returns an array as [<type>-Array =] rather than as [<type>-Var =]. For example:

```
[ Integer-Array = ] <com-handle>: retIntArray ( ).

[ Date-Array = ] <com-handle>: Dates.
```

You also cannot get or set the value of an individual COM array element without first storing the entire array into an ABL EXTENT variable. For example, each of the following will result in a runtime error:

```
dtXmas = ComObj:Holidays[2].

ComObj:Holidays[2] = dtXmas.

carModel = ComObj:Cars[1]:Model.
```

Pointer conversions

When indicated, ABL converts the specified ABL data type to a Pointer or Variant Pointer COM data type. This has no effect on the value of the parameter or property, only on the way the value is packaged. ABL determines the conversion in the following ways:

- Converts to a Pointer when the method parameter or property reference includes the BY-POINTER type option, or the Type Library specifies a Pointer to the corresponding COM data type
- Converts to a Variant Pointer when the method parameter or property reference includes the BY-VARIANT-POINTER type option, or the Type Library specifies a Variant Pointer to the corresponding COM data type

ABL data types not converted

ABL does not convert the BLOB, CLOB, MEMPTR, ROWID, or WIDGET-HANDLE data types to COM data types.

Alternate COM data type names

Documentation on COM objects generally specifies COM data type names similar to those shown in Table 75: Default conversions from COM data types to ABL data types on page 611 and Table 76: Conversions from ABL to COM data types on page 612. However, some documentation and some COM object viewers might use an alternative nomenclature. The following table shows the most common alternates.

Table 77: Alternative COM data type names

Alternative name	Common name
VT_ARRAY	Array
VT_BOOL	Boolean
VT_BSTR	String
VT_BYREF	Pointer
VT_BYREF + VT_VARIANT	Variant Pointer
VT_CY	Currency
VT_DATE	Date
VT_DISPATCH	Object
VT_ERROR	Error Code
VT_I1	Signed Byte
VT_I2	Signed Short (2-byte integer)
VT_I4	Signed Integer (4-byte integer)
VT_I8	Signed 8-byte Integer
VT_PTR	Pointer
VT_PTR + VT_VARIANT	Variant Pointer
VT_R4	Float (Single)
VT_R8	Double
VT_UI1	Unsigned Byte

Alternative name	Common name
VT_UI2	Unsigned Short (2-byte integer)
VT_UI4	Unsigned Integer (4-byte integer)
VT_UI8	Unsigned 8-byte Integer
VT_UNKNOWN	Object
VT_VARIANT	Variant (*anytype*)

B

Audit Policy Maintenance APIs

This appendix describes several ABL (Advanced Business Language) APIs used to implement the OpenEdge Audit Policy Maintenance tool. This tool allows you to create and maintain audit policies in an OpenEdge RDBMS. For more information on audit policies and the Audit Policy Maintenance tool, see *Introduction to Security and Auditing*.

If you want to provide a different front-end for the Audit Policy Maintenance tool, you use the information in this appendix to understand how to maintain the same back end. If you want to build your own auditing configuration tools, you can use the APIs described in this appendix as building blocks for these custom tools.

For basic information on implementing auditing tools in ABL, see Auditing on page 241.

For details, see the following topics:

- API overview
- Generic utility API
- Caching API
- Exporting policies to XML
- Importing policies from XML
- Additional audit policy procedures

API overview

The available APIs Audit Policy Maintenance consist of a set of persistent and non-persistent procedures. You can find these procedures and supporting files installed in the following OpenEdge location:

```
OpenEdge_install_dir/src/auditing/
```

The supporting files include the temp-table definitions (`src/auditing/ttdefs`) and ProDataSet definition (`src/auditing/include/dspolicy.i`) required by these APIs.

Note: This directory also contains the entire source code for the Audit Policy Maintenance tool, including the main utility procedure, `_apmt.p`.

Two of the APIs perform the same set of functions at different levels of generality:

- A generic utility API
- A caching API

Both APIs use ABL ProDataSets to read and save the audit configuration stored in metaschema tables of an OpenEdge RDBMS. The caching API uses a predefined ProDataSet instantiated by the Audit Policy Maintenance tool and implements audit configuration operations by passing this ProDataSet to the generic utility API. The generic utility API can directly implement the same audit configuration read and save operations using any ProDataSet that has the same definition as the Audit Policy Maintenance tool ProDataSet.

The remaining APIs consist of another persistent procedure and several standalone non-persistent procedures that perform a variety of specific functions. These APIs are also used by both the generic utility and caching APIs.

Reference to API procedure files

The following table lists all the persistent and non-persistent procedures that comprise these APIs, their purposes, and where you can find them documented in this appendix.

Table 78: Audit Policy Maintenance API procedure files

This procedure filein src\auditing...	Performs this function...	As described in the...
`_aud-utils.p`	Reads and saves audit configurations using a generic utility API (persistent)	Generic utility API on page 620
`_aud-cache.p`	Reads and saves audit configurations using a caching API (persistent)	Caching API on page 630
`_get-db-list.p`	Gets a list of connected databases with additional information about each database	Additional audit policy procedures on page 641

This procedure filein src\auditing...	Performs this function...	As described in the...
_exp-policies.p	Exports policies from a ProDataSet to an XML file	Exporting policies to XML on page 636
_exp-policies-db.p	Exports policies from a connected database to an XML file	Exporting policies to XML on page 636
_get-audevents.p	Fills the specified ProDataSet with the _aud-event table records from a specified database	Additional audit policy procedures on page 641
_aud-conflict.p	Either checks for audit policy conflicts or aggregates all active audit policies, returning a list of all data being audited	Additional audit policy procedures on page 641
_get-policies.p	Reads the audit policies from a specified database into the specified ProDataSet	Additional audit policy procedures on page 641
_imp-policies.p	Imports an XML file containing audit policies into either a database or a ProDataSet using an internal API (persistent)	Importing policies from XML on page 637
_update-policies.p	Saves to a database the tracked changes to policies stored in a ProDataSet	Additional audit policy procedures on page 641

Using the generic utility and caching APIs with an application server

The generic utility and caching APIs support execution on an application server. They provide functions to define the application server connection to be used when a local database is not present. The functions that read data from or write data to a database forward their requests to the application server whenever they detect that the database is not local. For application server execution, you can only call the procedures in the caching or generic utility APIs directly, as they are the only APIs that are application server aware.

If you want to call any of the other standalone procedures directly, your code needs to handle the forwarding of requests to the application server. For example, in the following combination of pseudo-code and ABL, the caller handles the application server connection:

```
DEFINE VARIABLE hAppSrv AS HANDLE    NO-UNDO.
DEFINE VARIABLE hDSet   AS HANDLE    NO-UNDO.
DEFINE VARIABLE cError  AS CHARACTER NO-UNDO.

CREATE SERVER hAppSrv.
hAppSrv:CONNECT("-AppService asbroker1 -H localhost- S 5162").

RUN auditing/_get-policies ON hAppSrv
  ("sports2000", OUTPUT DATASET-HANDLE hDSet, OUTPUT cError).
```

Generic utility API

The following functions and internal procedures are in `src/auditing/_aud-utils.p`. The procedures and functions herein described can be used by other applications if developers want to take advantage of this code and build a different UI or use their own ProDataSet object, for instance.

cancel-import-from-xml procedure

Deletes the persistent procedure started in policies-dataset-read-xml procedure on page 626 to load an XML file. This should only be called if the call to `policies-dataset-read-xml` returns a list of duplicate policy names and the caller decides not to call override-policies-from-xml procedure on page 626 to override the policies.

Parameters: None

Example: The following example is a combination of pseudo-code and ABL that shows how cancel-import-from-xml procedure on page 620, policies-dataset-read-xml procedure on page 626, and override-policies-from-xml procedure on page 626 can be used together.

Calling API procedures to import audit policies in XML

```
{auditing/include/_aud-utils.i}

RUN policies-dataset-read-xml IN hAuditUtils
  (INPUT pcxmlFileName, INPUT-OUTPUT DATASET-HANDLE hAuditDset BY-REFERENCE,
   OUTPUT pcList, OUTPUT pcErrorMsg).

/* Either display error message or confirmation that the policies were
   imported */
IF errorMsg <> "":U THEN
  MESSAGE errorMsg VIEW-AS ALERT-BOX ERROR.
ELSE DO:
  /* check if XML file has policies which already exist */
  IF cDupList <> "" THEN DO:
    MESSAGE "The following policies already exist:" SKIP
      REPLACE(cDupList,",",CHR(10)) SKIP
      "Do you want to override them?" SKIP
      "(If Yes, the listed policies will be deleted and re-imported)"
      VIEW-AS ALERT-BOX QUESTION BUTTON YES-NO UPDATE lChoice AS LOGICAL.
    IF lChoice THEN DO:
      /* User confirmed that he wants to override existing policies, so let's
         pick up from where we left off. Keep the changes. */
      RUN setcursor ("WAIT":U).
      RUN override-policies-from-xml
        (INPUT-OUTPUT DATASET-HANDLE hAuditDset BY-REFERENCE,
         OUTPUT pcErrorMsg).
      RUN setcursor ("":U).
      IF errorMsg <> "" THEN
        MESSAGE errorMsg VIEW-AS ALERT-BOX ERROR.
    END. /* lChoice */
    ELSE DO:
      /* User doesn't want to override policies, to cancel the previous
         request */
      RUN cancel-import-from-xml IN hAuditUtils.
      MESSAGE "Import canceled" VIEW-AS ALERT-BOX INFO.
    END.
  END.
END. /* errorMsg = "" */
```

check-conflicts procedure

Checks for conflicts on all policies stored in the ProDataSet object passed by the caller. This is a wrapper for `src/auditing/_aud-conflict.p`.

Parameters:

`INPUT DATASET-HANDLE FOR hDset`

ProDataSet object with policies.

`OUTPUT TABLE-HANDLE FOR ttHandle`

Temp-table handle were we store conflict information. The definition of the temp-table is in the `src/auditing/include/aud-report.i` include file.

export-audit-events procedure

Exports the audit events from the database specified to a `.ad` file. This routine retrieves the data by using a ProDataSet and then exports the data to the `.ad` file. see export-cached-audit-events procedure on page 621 if you want to export the information already cached in a ProDataSet and you have your own ProDataSet object.

Parameters:

`INPUT pcDbInfo AS CHARACTER`

Database information string.

`INPUT pcFileName AS CHARACTER`

Filename of the `.ad` file.

`OUTPUT pnumRecords AS INTEGER`

Number of records exported.

Note: Check for any error messages using the ABL `RETURN-VALUE` function.

---

export-cached-audit-events procedure

Exports the audit events from the ProDataSet specified to a `.ad` file. This routine reads the data cached in the ProDataSet object passed. see export-audit-events procedure on page 621 if you want to refresh the data before exporting.

Parameters:

`INPUT pcFileName AS CHARACTER`

Filename of the `.ad` file.

`INPUT DATASET-HANDLE phDataset`

ProDataSet object.

`OUTPUT pnumRecords AS INTEGER`

Number of records exported.

Note: Check for any error messages using the ABL `RETURN-VALUE` function.

fill-audit-dataset procedure

Fills a ProDataSet with the audit policies from the database specified.

Parameters:

`INPUT pcDbInfo AS CHARACTER`

Database information string that contains at least the logical database name to read the policies from.

`OUTPUT DATASET-HANDLE FOR phDset`

The ProDataSet object.

Note: Check for any error messages using the ABL `RETURN-VALUE` function.

Example:

```
{auditing/include/_aud-utils.i}

DEFINE VARIABLE hAuditDset AS HANDLE NO-UNDO.

RUN fill-audit-dataset IN hAuditUtils (ldbname(1),
  OUTPUT DATASET-HANDLE hAuditDset BY-REFERENCE).
```

getAuditEvents procedure

Populates a ProDataSet with the audit events from the specified database.

Parameters:

`INPUT pcDbInfo AS CHARACTER`

Database info string specifying the source database.

`INPUT plGetAllEvents AS LOGICAL`

Determines if caller also wants to get system events.

`OUTPUT DATASET-HANDLE FOR hEventDset`

The ProDataSet object.

get-complete-dbname-list procedure

Gets complete list of connected databases by calling `src/auditing/_get-db-list.p`. It also handles cases where the database is on the application server, if there is connection to the application server. The list contains entries separated by `CHR(1)`. Each entry contains the logical database name. It might also contain some additional information such as "`read-only`" or "`appserver`". The list of possible entries is defined in `src/auditing/include/_aud-std.i`.

Parameters:

`OUTPUT pcList AS CHARACTER`

List of databases, each database is separated by the value of `CHR(1)`.

Example:

```
{auditing/include/_aud-utils.i}

DEFINE VARIABLE cList   AS CHARACTER NO-UNDO.
DEFINE VARIABLE cdbInfo AS CHARACTER NO-UNDO.
DEFINE VARIABLE iLoop   AS INTEGER   NO-UNDO.

RUN get-complete-dbname-list IN hAuditUtils (OUTPUT cList).

DO iLoop = 1 TO NUM-ENTRIES(cList, CHR(1)):
  ASSIGN cdbInfo = ENTRY(iLoop, cList, CHR(1)).
END.
```

get-DB-Name function

Gets the database name out of the database information string returned by get-complete-dbname-list procedure on page 623, get-dbname-list procedure on page 624, and `src/auditing/_get-db-list.p`.

Returns: `CHARACTER`

Parameters:

`INPUT pcDbInfo AS CHARACTER`

Database information string.

Example:

```
{auditing/include/_aud-utils.i}

DEFINE VARIABLE cdbInfo AS CHARACTER NO-UNDO.
DEFINE VARIABLE cList   AS CHARACTER NO-UNDO.
DEFINE VARIABLE cTemp   AS CHARACTER NO-UNDO.
DEFINE VARIABLE iLoop   AS INTEGER   NO-UNDO.

RUN get-dbname-list IN hAuditUtils (OUTPUT cList).

DO iLoop = 1 TO NUM-ENTRIES(cList, CHR(1)):
  ASSIGN cdbInfo = ENTRY(iLoop, cList, CHR(1))
         cTemp   = DYNAMIC-FUNCTION('get-DB-Name' IN hAuditUtils, cdbInfo).
END.
```

get-dbname-list procedure

Gets list of connected databases with auditing enabled by calling `src/auditing/_get-db-list.p`. It also handles cases where the database is on the application server, if there is a connection to an application server. The list contains entries separated by the value of `CHR(1)`. Each entry contains the logical database name. It may also contain additional information such as `"read-only"` or `"appserver"`. List of possible entries is defined in `src/auditing/include/_aud-std.i`.

Parameters:

`OUTPUT pcList AS CHARACTER`

List of databases, each database separated by the value of `CHR(1)`.

Example:

```
{auditing/include/_aud-utils.i}

DEFINE VARIABLE cdbInfo AS CHARACTER NO-UNDO.
DEFINE VARIABLE cList   AS CHARACTER NO-UNDO.
DEFINE VARIABLE iLoop   AS INTEGER   NO-UNDO.

RUN get-dbname-list IN hAuditUtils (OUTPUT cList).

DO iLoop = 1 TO NUM-ENTRIES(cList, CHR(1)):
  ASSIGN cdbInfo = ENTRY(iLoop, cList, CHR(1)).
END.
```

get-policies-merge procedure

Gets the merged version of the active policies so the caller can generate a report of all settings that are turned on for auditing. This procedure reads only active audit policies.

Parameters:

`INPUT DATASET-HANDLE FOR hDset`

ProDataSet with the policies.

`OUTPUT TABLE-HANDLE FOR ttHandle`

Temp-table where we will store merged information. The definition of the temp-table is in the `src/auditing/include/aud-report.i` include file.

has-DB-option function

Looks for a given option in the database information string. The database information string is returned by get-complete-dbname-list procedure on page 623, get-dbname-list procedure on page 624, and `src/auditing/_get-db-list.p`. The list of possible entries is defined in `src/auditing/include/_aud-std.i`.

Returns: `LOGICAL`

Parameters:

`INPUT pcDbInfo AS CHARACTER`

The database information string.

`INPUT pcDbOption AS CHARACTER`

String containing the entry to look for.

Example:

```
{auditing/include/_aud-utils.i}

DEFINE VARIABLE cdbInfo AS CHARACTER NO-UNDO.
DEFINE VARIABLE cList   AS CHARACTER NO-UNDO.
DEFINE VARIABLE cTemp   AS CHARACTER NO-UNDO.
DEFINE VARIABLE iLoop   AS INTEGER   NO-UNDO.

RUN get-dbname-list IN hAuditUtils (OUTPUT cList).

DO iLoop = 1 TO NUM-ENTRIES(cList, CHR(1)):
  ASSIGN cdbInfo = ENTRY(iLoop, cList, CHR(1))
         cTemp   = DYNAMIC-FUNCTION('get-DB-Name' IN hAuditUtils, cdbInfo).
  IF DYNAMIC-FUNCTION('has-DB-option' IN hAuditUtils, cDbInfo,     "READ-ONLY":U) THEN
    MESSAGE "Database " cTemp " is read-only".
END.
```

isDBOnAppServer function

Checks if the database specified by a given database information string is on the application server. If we are running a WebClient session, this is always the case, so it will always return true for the WebClient. The database information string is returned by get-complete-dbname-list procedure on page 623, get-dbname-list procedure on page 624, and `src/auditing/_get-db-list.p`.

Returns: `LOGICAL`

Parameters:

`INPUT pcDbInfo AS CHARACTER`

Database information string.

Example:

```
{auditing/include/_aud-utils.i}

DEFINE VARIABLE cdbInfo AS CHARACTER NO-UNDO.
DEFINE VARIABLE cList   AS CHARACTER NO-UNDO.
DEFINE VARIABLE cTemp   AS CHARACTER NO-UNDO.
DEFINE VARIABLE iLoop   AS INTEGER   NO-UNDO.

RUN get-dbname-list IN hAuditUtils (OUTPUT cList).

DO iLoop = 1 TO NUM-ENTRIES(cList, CHR(1)):
  ASSIGN cdbInfo = ENTRY(iLoop, cList, CHR(1))
         cTemp   = DYNAMIC-FUNCTION('get-DB-Name' IN hAuditUtils, cdbInfo).
  IF DYNAMIC-FUNCTION('isDbOnAppServer' IN  hAuditUtils, cDbInfo) THEN
    MESSAGE "Database " cTemp " is not local".
END.
```

override-policies-from-xml procedure

This should only be called after a call to policies-dataset-read-xml procedure on page 626 which returns a list with duplicate policy names. By calling this procedure, the caller gets the data, which is loaded from the XML file into the specified ProDataSet object. The previous call to `policies-dataset-read-xml` keeps the data cached so that it doesn't have to re-read and parse the XML file again.

Parameters:

`INPUT-OUTPUT DATASET-HANDLE FOR phDataset`

Target ProDataSet to copy the data.

`OUTPUT pcErrorMsg AS CHARACTER`

Error messages.

Note: For more information, see policies-dataset-read-xml procedure on page 626.

Example: see cancel-import-from-xml procedure on page 620.

policies-dataset-read-xml procedure

Reads an XML file containing audit policies and then populate the specified ProDataSet object. If the ProDataSet already contains one or more policies defined in the XML file, this routine returns a comma-separated list of duplicate policies in the `pcList` parameter, so that caller can decide if it wants to override existing policies while loading the XML into the ProDataSet. This procedure calls `src/auditing/_imp-policies.p` to read the XML file and import it into the ProDataSet.

Parameters:

`INPUT pcxmlFileName AS CHARACTER`

XML file name.

`INPUT-OUPUT DATASET-HANDLE FOR phDataSet`

ProDataSet object.

`OUTPUT pcList AS CHARACTER`

List of existing policies.

`OUTPUT pcErrorMsg AS CHARACTER`

Error messages.

Example:

See cancel-import-from-xml procedure on page 620.

policies-dataset-write-xml procedure

Generates an XML file with policy settings stored in the specified ProDataSet object. This is a wrapper for `src/auditing/_exp-policies.p`.

Parameters:

`INPUT pcPolicyList AS CHARACTER`

Comma-separated list of policies to export, or `'*'` for all policies.

`INPUT pcxmlFileName AS CHARACTER`

Filename of generated XML file.

`INPUT DATASET-HANDLE FOR phDataSet`

ProDataset object holding the policy settings.

`OUTPUT pcErrorMsg AS CHARACTER`

Error messages.

setAppServerHandle procedure

Sets the server handle of an application server specified by the caller and uses it whenever this procedure needs to forward a request to the application server.

Parameters:

`INPUT phAppServer AS HANDLE`

Server handle for an application server connection.

updateAuditEvent procedure

Saves the changes tracked in the given ProDataSet object into the database specified by the database information string. The ProDataSet should at least contain a table with the audit events records. The caller should check the ProDataSet and buffer's ERROR attribute for any error conditions.

Parameters:

`INPUT pcDbInfo AS CHARACTER`

Database information string specifying the target database to save the changes.

`INPUT-OUTPUT DATASET-HANDLE phDataset`

ProDataSet with the tracked changes.

update-auditing-policies procedure

Saves the audit policy changes tracked in a specified ProDataSet object into a specified target database. The caller should check the ProDataSet and buffer's ERROR attribute for any error conditions and the buffer's ERROR-STRING for error messages.

Parameters:

`INPUT pcDbInfo AS CHARACTER`

Database information string specifying the target database to save the changes.

`INPUT-OUTPUT DATASET-HANDLE phDataset`

ProDataSet with the tracked changes.

Note: This procedure calls the `_update-policies.p` external procedure, which automatically refreshes the database cache by calling the `AUDIT-POLICY:REFRESH-AUDIT-POLICY()` method.

Example:

The following example is a combination of pseudo-code and ABL that shows how to run this procedure:

Calling the update-auditing-policies API procedure

```
{auditing/include/_aud-utils.i}

DEFINE VARIABLE cStatus    AS CHARACTER NO-UNDO.
DEFINE VARIABLE errorMsg   AS CHARACTER NO-UNDO.
DEFINE VARIABLE hBuffer    AS HANDLE    NO-UNDO.
DEFINE VARIABLE hDSChanges AS HANDLE    NO-UNDO.
DEFINE VARIABLE hQuery     AS HANDLE    NO-UNDO.
DEFINE VARIABLE ix         AS INTEGER   NO-UNDO.

/* Turn tracking-changes off for now */
RUN set-tracking-changes (FALSE).

/* Create a ProDataSet with the changes before sending it to the other
   procedure */
CREATE DATASET hDSChanges.
hDSChanges:CREATE-LIKE(hAuditDset).
hDSChanges:GET-CHANGES(hAuditDset). /* Get the changes */
ASSIGN errorMsg = "".

/* Call the generic procedure passing our local ProDataSet */
RUN update-auditing-policies IN hAuditUtils (INPUT cDbInfo,
  INPUT-OUTPUT DATASET-HANDLE hDSChanges BY-REFERENCE).
ASSIGN errorMsg = RETURN-VALUE.

/* Turn tracking changes back on */
RUN set-tracking-changes (TRUE).

/* Check the ERROR status that might have been returned. */
IF errorMsg = "":U AND hDSChanges:ERROR THEN DO:
  /* There was an error somewhere in the updates. Find it. */
  DO ix = 1 TO hDSChanges:NUM-BUFFERS.
    CREATE QUERY hQuery.
    hBuffer = hDSChanges:GET-BUFFER-HANDLE(ix):BEFORE-BUFFER.
    hQuery:ADD-BUFFER(hBuffer).
    hQuery:QUERY-PREPARE("FOR EACH " + hBuffer:NAME).
    hQuery:QUERY-OPEN().
    hQuery:GET-FIRST().

    DO WHILE NOT hQuery:QUERY-OFF-END:
      IF hBuffer:ERROR THEN
        cStatus = cStatus + hBuffer:ERROR-STRING + CHR(10) NO-ERROR.
      hQuery:GET-NEXT().
    END.
    hQuery:QUERY-CLOSE().
    DELETE OBJECT hQuery.
  END.
END.

/* delete the ProDataSet with changes */
DELETE OBJECT hDSChanges.
IF errorMsg <> "":U THEN
  RETURN errorMsg.

IF cstatus = "" THEN
  /* If no errors so far, accept all the changes */
  hAuditDset:ACCEPT-CHANGES().
ELSE
  RETURN cStatus.
```

Caching API

The procedure `src/auditing/_aud-cache.p` contains the API for caching the auditing policies from a given database into a ProDataSet object. The defined procedures and functions can be used by other applications if you want to use this API, for example, to build a different UI.

The procedures take advantage of ProDataSets to read and write policies from and to databases. This procedure works as a sort of wrapper for the API defined in `src/auditing/_aud-utils.p`. This procedure creates a ProDataSet object (with temp-tables defined by the caller) and maintains it for the caller so the caller does not need to build one.

The following example is a combination of pseudo-code and ABL that shows how to use the caching API described in the following sections:

Example calls to the audit policy caching API

```
{auditing/include/_aud-cache.i}

/* Define the temp-tables */
{auditing/ttdefs/ _audpolicytt.i}
{auditing/ttdefs/_audfilepolicytt.i}
{auditing/ttdefs/_audfieldpolicytt.i}
{auditing/ttdefs/_audeventpolicytt.i}

/* Call either registerAuditTableHandle for each table or
   set-tt-hdls-for-dataset to pass the four table handles */

/*
RUN registerAuditTableHandle IN hAuditCacheMgr ("policy",
  TABLE ttAuditPolicy:HANDLE).
RUN registerAuditTableHandle IN hAuditCacheMgr ("file-policy",
  TABLE ttAuditFilePolicy:HANDLE).
RUN registerAuditTableHandle IN hAuditCacheMgr ("field-policy",
  TABLE ttAuditFieldPolicy:HANDLE).
RUN registerAuditTableHandle IN hAuditCacheMgr ("event-policy",
  TABLE ttAuditFieldPolicy:HANDLE).
*/

RUN set-tt-hdls-for-dataset IN hAuditCacheMgr(TABLE ttAuditPolicy:HANDLE,
  TABLE ttAuditFilePolicy:HANDLE, TABLE ttAuditFieldPolicy:HANDLE,
  TABLE ttAuditEvent Policy:HANDLE).

/* Populate temp-tables with policies from connected database */
RUN changeAuditDatabase IN hAuditCacheMgr (ldbname(1)).

/* load policies from XML file */
RUN import-policies-from-xml IN hAuditCacheMgr
  (INPUT cFileName, OUTPUT cDupList, OUTPUT errorMsg).

/* Either display error message or confirmation that the policies were
   imported */
IF errorMsg <> "":U THEN
  RETURN ERROR errorMsg.
ELSE DO:
/* Check if XML file has policies which already exist */
IF cDupList <> "" THEN DO:
  MESSAGE "The following policies already exist:" SKIP
    REPLACE(cDupList,",",CHR(10)) SKIP
    "Do you want to override them?" SKIP
    "(If Yes, the listed policies will be deleted and re-imported)"
    VIEW-AS ALERT-BOX QUESTION BUTTON YES-NO UPDATE lChoice AS LOGICAL.

  IF lChoice THEN DO:
```

```
    /* User confirmed that he wants to override existing policies, so let's
       pick up from where we left off. Keep the changes. */
    RUN setcursor ("WAIT":U).
    RUN resubmit-import-from-xml IN hAuditCacheMgr(OUTPUT errorMsg).
    RUN setcursor ("":U).
    IF errorMsg <> "" THEN
      RETURN ERROR errorMsg.
  END.

  ELSE DO:
    /* User doesn't want to override policies, so cancel the previous
       request */
    RUN cancel-import-from-xml IN hAuditCacheMgr.
    MESSAGE "Import canceled" VIEW-AS ALERT-BOX INFO.
  END.
END. /* cDupList <> "" */

/* Make some changes to the policies by updating the temp-tables. Check if
   there are pending changes and save them to the database */
RUN getDataModified IN hAuditCacheMgr (OUTPUT lMod).
IF lMod THEN DO: /* changes pending */
  RUN saveChangesAuditDatabase IN hAuditCacheMgr.
  ASSIGN errorMsg = RETURN-VALUE.
  IF errorMsg <> "":U THEN DO:
    /* Let's reject everything and get out */
    RUN rejectChangesAuditDatabase IN hAuditCacheMgr.
    RETURN ERROR errorMsg.
  END.

  /* Export changed policies to an XML file. Call the procedure in the
     auditUtils procedure to do the job */
  RUN export-policies-to-xml IN hAuditCacheMgr
    (INPUT cList, INPUT cFileName, OUTPUT errorMsg).
  IF errorMsg <> "":U THEN
    RETURN ERROR errorMsg.
END.
```

changeAuditDatabase procedure

This procedure fills the internal ProDataSet object with the policy configuration from the database specified in a database information string. This should only get executed if the caller does not have its own ProDataSet object and wants this procedure to maintain one for them. If the caller passes an empty string, this procedure empties the temp-tables in the ProDataSet.

Parameters:

`INPUT pcDbInfo AS CHARACTER`

Database information string specifying the database with policy configuration information.

Note: Check for any error messages using the ABL `RETURN-VALUE` function.

---

export-cached-audit-events procedure

Exports the audit events cached to a `.ad` file. This routine retrieves the data cached in the local ProDataSet object. See `exportAuditEvents` in `src/auditing/_aud-utils.p` if you want to refresh the data before exporting.

Parameters:

`INPUT pcFileName AS CHARACTER`

Filename of the .ad file to generate.

`OUTPUT pnumRecords AS INTEGER`

Number of records exported.

Note: Check for any error messages using the ABL `RETURN-VALUE` function.

---

export-policies-to-xml procedure

Exports policies stored in the local ProDataSet to an XML file. The caller specifies the list of policies to be exported, which could be "*" to export all policies stored in the ProDataSet.

Parameters:

`INPUT pcPolicyList AS CHARACTER`

A comma-separated list of policies to export.

`INPUT pcxmlFileName AS CHARACTER`

Filename of the XML file to generate.

`OUTPUT pcErroMsg AS CHARACTER`

Error messages.

get-audit-events procedure

Populates the internal temp-table with the records from the `_aud-event` table in the specified database.

Parameters:

`INPUT pcDbInfo AS CHARACTER`

Database information string or a logical database name.

get-conflict-info procedure

Calls the generic function to check for conflicts on all active policies stored in the internal ProDataSet.

Parameters:

`OUTPUT TABLE-HANDLE FOR ttHandle`

Temp-table to store the information.

`OUTPUT pcErrorMsg AS CHARAFCTER`

Error messages.

getDataModified procedure

Checks if there are any pending changes in the local ProDataSet object (populated by changeAuditDatabase procedure on page 631) which have not been saved.

Parameters:

`OUTPUT lMod AS LOGICAL`

YES if there are changes pending, otherwise NO.

get-merge-info procedure

Calls the generic function to get a report of the merged policies stored in the internal ProDataSet.

Parameters:

`OUTPUT TABLE-HANDLE FOR ttHandle`

Temp-table to store the report data.

`OUTPUT pcErrorMsg AS CHARACTER`

Error messages.

getNextAppLevelEventID function

Finds the last application level event ID in the local temp-table storing the audit event records and return the next value.

Returns: CHARACTER

Parameters: None

hasApplEvents function

Returns YES if there are application-level events defined in the local temp-table storing the audit event records.

Returns: LOGICAL

Parameters: None

import-audit-events procedure

Imports audit events from a .ad file into the local ProDataSet. You must call saveAuditEventChanges procedure on page 635 to get changes saved into the database.

Parameters:

`INPUT pcFileName AS CHARACTER`

Filename of the `.ad` file to import.

`INPUT perror% AS INTEGER`

Percentage of errors tolerated (similar to Data Administration database load process). Specifying 0 means any error aborts the import process, and specifying 100 means continue importing audit events no matter how many errors are found.

`OUTPUT pnumRecords AS INTEGER`

Number of records imported.

import-policies-from-xml procedure

Imports policies from a specified XML file into the local ProDataSet object. If the XML file contains duplicate policy names, this procedure returns a comma-separated list of duplicate policy names, in which case the caller can decide if it wants to override them and call resubmit-import-from-xml procedure on page 635.

Parameters:

`INPUT pcxmlFileName AS CHARACTER`

Filename of the XML file to import from.

`OUTPUT pcList AS CHARACTER`

List of existing policies.

`OUTPUT pcErrorMsg AS CHARACTER`

Error messages.

is-valid-event-id function

Verifies if the specified event ID is valid. It checks the cached information in the local temp-table storing the audit event record.

Returns: `LOGICAL`

Parameters:

`INPUT pEvent-id AS INTEGER`

Event ID number.

registerAuditTableHandle procedure

Caller specifies the handle of a temp-table that this procedure uses when building a local ProDataSet object on its behalf. Note that the temp-table definitions used by the Audit Policy Maintenance tool are located in `src/auditing/ttdefs`.

Parameters:

`INPUT pcID AS CHARACTER`

Identifier for the table, which can be `"policy"`, `"file-policy"`, `"field-policy"`, or `"event-policy"`.

`INPUT TABLE-HANDLE FOR hTable`

Handle of the temp-table.

rejectChangesAuditDatabase procedure

Rejects the changes tracked in the local ProDataSet object (containing the audit policies), undoing any changes to the records in the local database object.

Parameters: None

resubmit-import-from-xml procedure

Overrides any existing policy with the policies define in the XML file. If you want to overwrite the duplicate policies, you should call this procedure after a call to import-policies-from-xml procedure on page 634, which returns a list of duplicate policies.

Parameters:

`OUTPUT pcErrorMsg AS CHARACTER`

Error messages.

saveAuditEventChanges procedure

Saves the changes tracked in the local temp-table that holds the audit event records into the same database where the records are read from.

Parameters: None

Note: Check for any error messages using the ABL `RETURN-VALUE` function.

---

saveChangesAuditDatabase procedure

Saves changes tracked in the local ProDataSet object (containing the audit policies) into the same database where the policies are read from in changeAuditDatabase procedure on page 631.

Parameters: None

Note:

- Check for any error messages using the ABL `RETURN-VALUE` function.
- This procedure calls update-auditing-policies procedure on page 628 in the generic utility API.

---

set-tt-hdls-for-dataset procedure

The caller specifies the handles of temp-tables that this procedure uses when building a local ProDataSet object on its behalf. Note that the temp-table definitions used by the Audit Policy Maintenance tool are in `src/auditing/ttdefs`.

Parameters:

`INPUT pcList AS CHARACTER`

Comma-separated list of handles. The handles must be specified in order of policy, file-policy, field-policy, and event-policy tables.

Exporting policies to XML

OpenEdge provides two external procedures in the `src\auditing` directory that allow you to export audit policies from a specified ProDataSet or connected database.

_exp-policies.p

Exports policies from a ProDataSet to an XML file. The caller passes a comma-separated list of policies or "*" for all, and the name of the XML file to be created. This procedure is used by both the Audit Policy Maintenance tool and the Data Administration tool when exporting audit policies to an XML file.

Parameters:

`INPUT pcPolicyList AS CHARACTER`

Comma-separated list of policies.

`INPUT pcxmlFileName AS CHARACTER`

Filename of the XML file to generate.

`INPUT DATASET-HANDLE phDataset`

ProDataSet object containing the policies.

`OUTPUT pcErroMsg AS CHARACTER`

Error messages.

_exp-policies-db.p

Exports policies from a connected database to an XML file. The caller passes a comma-separated list of policies or "*" for all, and the name of the XML file to be created.

Parameters:

`INPUT pcPolicyList AS CHARACTER`

Comma-separated list of policies.

`INPUT pcxmlFileName AS CHARACTER`

Filename of the XML file to generate.

`INPUT pcDbName AS CHARACTER`

Logical name of the connected database containing the policies.

`OUTPUT pcErroMsg AS CHARACTER`

Error messages.

Importing policies from XML

OpenEdge provides the persistent procedure, `src/auditing/_imp-policies.p` that provides an API to import an XML file containing audit policies into either a database or a ProDataSet. This procedure must be executed persistently and then the caller should execute the internal procedures depending on the target for the policies—a ProDataSet or a database. This procedure is used by both the Audit Policy Maintenance tool and the Data Administration tool when importing audit policies from an XML file.

For a database, call import-xml-db procedure on page 639 and save-changes-to-db procedure on page 640. For a ProDataSet, call import-xml-fill-dset procedure on page 639 and copy-changes-to-dset procedure on page 638. The `cleanup` internal procedure should be called before deleting the persistent procedure.

Note: Import procedure for loading audit policies from an XML file into an OpenEdge database does not refresh the database cache automatically. It's up to the caller to decide if the cache should be refreshed. You can either call refresh-db-cache procedure on page 640 or directly call the `AUDIT-POLICY:REFRESH-AUDIT-POLICY()` method.

The following example is a combination of pseudo-code and ABL that shows how to use some of the procedures that are provided in `_imp-policies.p` and described in the sections that follow:

Example calls to the API for importing audit policies from XML

```
DEFINE VARIABLE cErrorMsg AS CHARACTER NO-UNDO.
DEFINE VARIABLE cList     AS CHARACTER NO-UNDO.
DEFINE VARIABLE hProc     AS HANDLE    NO-UNDO.

/* Start the importing procedure */
RUN auditing/_imp-policies.p PERSISTENT SET hProc.

/* Try to load the file and save the changes into the database */
RUN import-xml-db IN hProc (INPUT "text.xml", /* XML file name */
  INPUT "sports2000", /* logical db name */
  INPUT TRUE,         /* wait on lock */
  INPUT FALSE,        /* override existing policies */
  OUTPUT cList,       /* list of duplicate policies */
  OUTPUT cErrorMsg).

/* Check if there was an error */
IF cErrorMsg = "":U THEN DO:
  IF cList <> "":U THEN DO:
    MESSAGE "The following policies already exist: " cList SKIP
      "Do you want to override them? (If you answer yes, the tool will" SKIP
      "delete the listed policies before importing them)"
      VIEW-AS ALERT-BOX QUESTION BUTTON YES-NO UPDATE lChoice AS LOGICAL.
    /* If the user wants to override policies, need to call save-changes-to-db
       to get them saved */
    IF lChoice THEN DO:
      RUN save-changes-to-db IN hProc
        (INPUT "sports2000", /* logical db name */
         INPUT TRUE,         /* wait on lock */
         OUTPUT cErrorMsg).
      IF cErrorMsg <> "" THEN  /* an error occurred */
        MESSAGE cErrorMsg VIEW-AS ALERT-BOX.
    END.
  END.
END.

ELSE /* An error occurred */
  MESSAGE cErrorMsg VIEW-AS ALERT-BOX ERROR.

/* Clean up the procedure and delete it */
RUN cleanup IN hProc.
DELETE PROCEDURE hProc.
```

copy-changes-to-dset procedure

Fills the specified ProDataSet with the records loaded with import-xml-fill-dset procedure on page 639. It's the caller's responsibility to reject the changes if an error occurs. If `import-xml-fill-dset` doesn't return duplicate policies or the caller specifies that it should override existing policies, `import-xml-fill-dset` automatically fills the ProDataSet with the records loaded, and this procedure should not be called.

Parameters:

`INPUT-OUTPUT DATASET FOR dsAudPolicy`

Target ProDataSet to copy the records.

`OUTPUT perrorMsg AS CHARACTER`

Error messages.

import-xml-db procedure

Imports an XML file containing audit polices into a connected database specified by its logical name. If the XML file contains policies that already exist in the database and the `ploverride` parameter is set to `NO`, this procedure does not update the database and returns a comma-separated list of existing policies. The caller can then ask for confirmation and call save-changes-to-db procedure on page 640. If `ploverride` is set to `YES`, this procedure automatically overrides all existing policies and does not return any duplicates in the `pcDupList` parameter. Any errors set the `perrorMsg` parameter and cause the import operation to be backed out.

Parameters:

`INPUT pcxmlFileName AS CHARACTER`

Filename of the XML file with audit policies to import.

`INPUT pcDbName AS CHARACTER`

Logical name of the target database to import the policies.

`INPUT plWait AS LOGICAL`

`YES` specifies to wait if a lock on a record cannot be acquired.

`INPUT ploverride AS LOGICAL`

`YES` specifies to override existing policies.

`OUTPUT pcDupList AS CHARACTER`

Comma-separated list of policies that already exist in the specified database and also exist in the XML file.

`OUTPUT perrorMsg AS CHARACTER`

Error messages.

import-xml-fill-dset procedure

Imports an XML file containing audit polices into a specified ProDataSet. If the XML file contains policies that already exist in the ProDataSet and the `ploverride` parameter is set to `NO`, this procedure does not fill the ProDataSet and returns a comma-separated list of existing policies. The caller can then ask for confirmation and call copy-changes-to-dset procedure on page 638. If `ploverride` is set to `YES`, this procedure automatically overrides all existing policies and does not return any duplicates in the `pcDupList` parameter. Any errors set the `perrorMsg` parameter. It is the caller's responsibility to reject any changes if errors occur.

Parameters:

`INPUT pcxmlFileName AS CHARACTER`

Filename of the XML file to import.

`INPUT ploverride AS LOGICAL`

`YES` specifies to override existing policies.

`INPUT-OUTPUT DATASET FOR dsAudPolicy`

Target ProDataSet to import the policies.

`OUTPUT pcDupList AS CHARACTER`

Comma-separated list of policies that already exist in the specified database and also exist in the XML file.

`OUTPUT perrorMsg AS CHARACTER`

Error messages.

refresh-db-cache procedure

Calls the method that refreshes the database cache with the newly loaded policy settings. This should only be called after a successful call has been made to import-xml-db procedure on page 639 or save-changes-to-db procedure on page 640. It is up to the caller to decide if the cache should be refreshed after new policy settings are loaded. See import-xml-db procedure on page 639 and save-changes-to-db procedure on page 640 for more information.

Parameters:

`INPUT pcDbName AS CHARACTER`

Logical name of the database to have its cache refreshed.

`OUTPUT perrorMsg AS CHARACTER`

Error messages.

save-changes-to-db procedure

Saves the policies previously loaded by import-xml-db procedure on page 639 into the specified database. This should **only** be called if `import-xml-db` returns a list of duplicate policies. If `import-xml-db` does not return duplicate policies or the caller specifies that it should override existing policies, `import-xml-db` automatically saves the changes into the database, and this procedure should not be called.

Parameters:

`INPUT pcDbName AS CHARACTER`

Logical name of the database in which to save the previously loaded policies.

`INPUT plWait AS LOGICAL`

YES specifies to wait if a lock on a record cannot be acquired.

`OUTPUT perrorMsg AS CHARACTER`

Error messages. Any error message causes the transaction to be backed out.

Additional audit policy procedures

These are some additional procedures used by one of the APIs described in previous sections. They can be called directly as long as you pass the right parameters.

_aud-conflict.p

Either checks for conflicts on active policies in the specified ProDataSet object or aggregates all active policies to get the effective settings (so you can get a list of everything to be audited). This procedure follows the rules for resolving conflicts specified by the auditing framework. This procedure populates a temp-table (defined in `src/auditing/include/aud-report.i`) with the information. Each line represents a table, field, or event setting. See the procedure code for more information on the format of the information.

Parameters:

`INPUT lMode AS LOGICAL`

YES to specify conflict checking mode, or `NO` to specify aggregating mode.

`INPUT DATASET-HANDLE FOR phDataset`

ProDataSet containing the policies to be checked or aggregated.

`INPUT-OUTPUT TABLE-HANDLE FOR ttHandle`

Handle to temp-table where the information is stored.

_get-audevents.p

Fills the specified ProDataSet with the `_aud-event` table records from the specified database.

Parameters:

`INPUT pcDbName AS CHARACTER`

Logical name of the source database.

`INPUT plGetAllEvents AS LOGICAL`

YES specifies to copy system events.

`OUTPUT DATASET-HANDLE phDataset`

ProDataSet object to fill with records from the `_aud-events` table.

`OUTPUT errorMsg AS CHARACTER`

Error messages.

_get-db-list.p

Gets a list of databases connected with some additional information about each database. The caller determines if the list should contain only audit-enabled databases. The list of possible entries is defined in `src/auditing/include/_aud-std.i`.

Parameters:

`INPUT pcgetCompleteList AS LOGICAL`

YES specifies to return a complete list of connected databases, including the nonaudit-enabled ones. If the user does not have permission (Audit Data Reporter) to read the policy tables for a database, the database is not included when the caller specifies `NO` to return a list of audit-enabled databases only.

`OUTPUT cList AS CHARACTER`

Returned list of databases. The information for each database is separated by the value of `CHR(1)`.

_get-policies.p

Reads the audit policy settings stored in a specified source database into a specified target ProDataSet object.

Parameters:

`INPUT pcDbName AS CHARACTER`

Logical name of the source database.

`OUTPUT DATASET-HANDLE phDataset`

ProDataSet object where the procedure stores the policies.

`OUTPUT erroMsg AS CHARACTER`

Error messages.

_update-policies.p

Saves the tracked changes to policies stored in a specified source ProDataSet into a specified target database. If an error occurs for any record update, the procedure sets the `ERROR` attribute on both the ProDataSet and buffer, as well as the buffer `ERROR-STRING` attribute. The caller should check for errors.

Parameters:

`INPUT pcDbName AS CHARACTER`

Logical name of the target database.

`INPUT-OUTPUT DATASET-HANDLE phDataset`

ProDataSet object containing the tracked changes to save.

`OUTPUT erroMsg AS CHARACTER`

Error messages.

Note: This procedure automatically refreshes the database cache by calling the `AUDIT-POLICY:REFRESH-AUDIT-POLICY()` method.

C

Change Data Capture ABL references

OpenEdge provides an ABL API to access and manage the Change Data Capture (CDC) schema information. The purpose of the API is to provide access to this schema information and allow the ABL code to perform typical operation such as create, read, edit, and delete for configuring CDC table and field policies.

Before using this API, you should be familiar with configuring table and field policies using the OpenEdge Data Dictionary, Database Administration Console in the OpenEdge Explorer and OpenEdge Management, and other database utilities.

Using the API, you can

- Create, read, edit, and delete CDC table policies.
- Create, read, edit, and delete CDC field policies.
- Create objects for schema tables.

The API uses a standard naming convention `I`*`ClassName`*`[Element|Collection]` for public interface names.

For details, see the following topics:

- API entities and entity service methods
- Code examples
- Enums
- Interface properties (Table policy)
- Interface properties (Field policy)
- API to generate Change Data Capture policy program

- ABL procedure to dump
- ABL procedure to load
- ABL support classes for CDC

API entities and entity service methods

The `ICdcTablePolicy` interface provides service methods that implement create, read, update, and delete operations against single schema element, or a collection of schema elements, accessed through supported entity and entity collection interfaces.

The available service methods provided by the API for creating and accessing the basic entities and entity collections are listed in the following topics:

- New( ) method (Table policy) on page 646
- New( ) method (Field policy) on page 647
- Create( ) method on page 647
- Get( ) method on page 648
- Update( ) method (Table policy) on page 649
- Delete( ) method on page 650

The listed entity name is the implementing class name for each entity object and is also part of each entity method name, for example, "CDC Table Policy" as in `NewCdcTablePolicy( CHARACTER )` method. Corresponding entity collection methods have a similar method name using the plural of the entity name as in `NewCdcTablePolicies( )`method.

New( ) method (Table policy)

Factory method to create a CDC table policy. Calling this method returns a new object for `ICDCTablePolicy` instance with `Name` as the customary parameter. The `Name` property of the `ICDCTablePolicy` object is assigned to the `_policy-name` field in the database and requires `Name` as the input parameter.

The returned `ICDCTablePolicy` object can be modified before creating the table policy in the database using the DataAdminService.

Return type: ICDCTablePolicy

Access: PUBLIC

Applies to: OpenEdge.DataAdmin.DataAdminService class

Syntax

```
NewCDCTablePolicy( INPUT policyName AS CHARACTER )
```

New( ) method (Table policies)

Factory method to create a collection object for CDC table policies. This method has no parameters and returns a new empty collection object. Add `ICdcTablePolicy` instances that are returned from the entity factory methods to the collection object and pass it to the corresponding `Create( )` method for the collection to commit all the entities to the database in a single transaction.

Return type: ICDCTablePolicySet

Access: PUBLIC

Applies to: OpenEdge.DataAdmin.DataAdminService class

Syntax

```
NewCDCTablePolicies( )
```

New( ) method (Field policy)

Adds a CDC field policy to a CDC table policy. Calling this method adds a new `ICDCFieldPolicy` to the `FieldPolicies` collection of a new or an existing `ICDCTablePolicy`. There is no factory method which returns an empty collection as the `FieldPolicies` collection always exist in the table policy.

Return type: ICDCFieldPolicy

Access: PUBLIC

Applies to: OpenEdge.DataAdmin.DataAdminService class

Syntax

```
NewCDCFieldPolicy( )
```

Create( ) method

Creates and commits the CDC table policy records to the database. This method requires an instance of `ICDCTablePolicy` object as the input parameter to create necessary database objects. This method returns TRUE on successful creation of a policy and returns FALSE on failure. Before calling this method, ensure that the assignments to the `ICDCTablePolicy` object are complete.

Return type: LOGICAL

Access: PUBLIC

Applies to: OpenEdge.DataAdmin.DataAdminService class

Syntax

```
CreateCDCTablePolicy( INPUT policy AS ICDCTablePolicy )
```

Create( ) method (Table policies)

Creates and commits CDC table policy records to the database. This method requires `ICDCTablePolicySet` collection object as the input parameter to create necessary database objects. This method creates multiple policies in a single transaction.

Return type: LOGICAL

Access: PUBLIC

Applies to: OpenEdge.DataAdmin.DataAdminService class

Syntax

```
CreateCDCTablePolicies( INPUT policySet AS ICDCTablePolicySet )
```

Get( ) method

Reads a CDC table policy and returns a `ICdcTablePolicy` object instance based on the policy name provided as input.

Return type: ICDCTablePolicy

Access: PUBLIC

Applies to: OpenEdge.DataAdmin.DataAdminService class

Syntax

```
GetCDCTablePolicy( INPUT policyName AS CHARACTER )
```

Get( ) method (Table policies)

Retrieves all CDC policies available for the selected database. This method returns `ICdcTablePolicySet` collection object

Return type: ICdcTablePolicySet

Access: PUBLIC

Applies to: OpenEdge.DataAdmin.DataAdminService class

Syntax

```
GetCdcTablePolicies( )
```

GetTable( ) method (Table policy)

Retrieves an object representing the table based on the input provided. This method supports both PUB and non-PUB, and has two parameters: `TableOwner` and `OwnerName`.

The `TableOwner` parameter takes the name of the table that belongs to PUB or non-PUB owner, and the `OwnerName` parameter takes the owner's name of the table. If the `OwnerName` parameter is left blank, the `Get( )` request fetches the table that belongs to PUB owner by default

Return type: ITable

Access: PUBLIC

Applies to: OpenEdge.DataAdmin.DataAdminService class

Syntax

```
GetTable( INPUT Name AS CHARACTER, INPUT Owner AS CHARACTER )
```

GetFilter( ) method (Table policies)

Retrieves CDC policies based on the filters applied. These filters can be Name, State, Instance, Level, or TableName. Consider the following example to use filters:

```
GetCdcTablePolicies("TableName=~"Customer~" and Instance=~"0~"")
```

Return type: ICdcTablePolicySet

Access: PUBLIC

Applies to: OpenEdge.DataAdmin.DataAdminService class

Syntax

```
GetCdcTablePolicies( INPUT filter AS CHARACTER)
```

Update( ) method (Table policy)

Updates a CDC table policy and its field policies. This method requires `ICdcTablePolicy` object as input parameter. Each `Get( )` method returns a single object instance and passes it to the corresponding `Update( )` method to commit to the database after editing in ABL code.

Before calling this method, retrieve the policy using `GetCDCTablePolicy( )` method with policy name as the input parameter and update the policy information with required changes.

Return type: LOGICAL

Access: PUBLIC

Applies to: OpenEdge.DataAdmin.DataAdminService class

Syntax

```
UpdateCDCTablePolicy( INPUT policy AS ICDCTablePolicy )
```

Update( ) method (Table policies)

Updates a CDC table policy collection and its field policies. This method requires `ICdcTablePolicySet` collection object as the input parameter to update a set of CDC Table policies. This method updates the policies in a single transaction.

Before calling this method, retrieve the policies using `GetCdcTablePolicies( )` method and update the information of policies with required changes.

Return type: LOGICAL

Access: PUBLIC

Applies to: OpenEdge.DataAdmin.DataAdminService class

Syntax

```
UpdateCDCTablePolicies( INPUT policySet AS ICDCTablePolicySet )
```

Delete( ) method

Deletes a CDC table policy and its field policies. Deleting a CDC table policy deletes all the field policies associated with the table policy from the database. The method requires `Name` as the input parameter to delete the CDC table policy from the database. If the CDC table policy with the specified policy name does not exist, the method will throw an error.

Return type: LOGICAL

Access: PUBLIC

Applies to: OpenEdge.DataAdmin.DataAdminService class

Syntax

```
DeleteCDCTablePolicy( INPUT policyName AS CHARACTER )
```

Code examples

For code examples, see the following topics:

- Create CDC policies on page 651
- Get CDC policies on page 654
- Update CDC policies on page 655
- Delete CDC policies on page 656

Create CDC policies

The following code is an example for creating CDC table policies:

```
using OpenEdge.DataAdmin.Error.DataAdminErrorHandler from propath.
using OpenEdge.DataAdmin.ICdcTablePolicy from propath.
using OpenEdge.DataAdmin.ICdcFieldPolicy from propath.
using OpenEdge.DataAdmin.ITable from propath.
using OpenEdge.DataAdmin.CdcTablePolicyStateEnum.
using OpenEdge.DataAdmin.CdcTablePolicyLevelEnum.

define variable service as OpenEdge.DataAdmin.DataAdminService no-undo.
define variable tablePolicy as ICdcTablePolicy no-undo.
define variable fieldPolicy as ICdcFieldPolicy no-undo.
define variable tbl as ITable no-undo.

service = new OpenEdge.DataAdmin.DataAdminService(ldbname(1)).

tablepolicy = service:NewCdcTablePolicy("test").
tbl = service:GetTable("customer","PUB").
assign
    tablePolicy:Table                   = tbl
    tablePolicy:Level                   = CdcTablePolicyLevelEnum:MAXIMUM
    tablePolicy:State                   = CdcTablePolicyStateEnum:Active
    tablePolicy:IdentifyingField        = YES
    tablePolicy:EncryptPolicy           = no
    tablePolicy:ChangeTable             = "changeTable"
    tablePolicy:ChangeTableOwner        = "PUB"
    tablePolicy:DataArea                = service:GetArea("customer index area")
    tablePolicy:IndexArea               = service:GetArea("customer index area")
    tablePolicy:Description             = "desc"
    .

fieldPolicy = service:NewCdcFieldPolicy().
tablePolicy:FieldPolicies:add(fieldPolicy).
fieldPolicy:Field = tbl:Fields:Find("name").
fieldPolicy:IdentifyingField = 1.

fieldPolicy = service:NewCdcFieldPolicy().
tablePolicy:FieldPolicies:add(fieldPolicy).
fieldPolicy:Field = tbl:Fields:Find("city").
fieldPolicy:IdentifyingField = 2.

service:CreateCdcTablePolicy(tablePolicy).
```

Create bulk CDC policies

The following code is an example for creating CDC table policies in bulk:

```
using OpenEdge.DataAdmin.* from propath.

define variable service          as DataAdminService   no-undo.
define variable myCdcTablePolicy as ICdcTablePolicy    no-undo.
define variable myCdcFieldPolicy as ICdcFieldPolicy    no-undo.
define variable tbl              as ITable             no-undo.
define variable tablePolicySet   as ICdcTablePolicySet no-undo.

service = new DataAdminService(ldbname(1)).

/* this is used to create collection of policies */
tablePolicySet = service:NewCdcTablePolicies().

myCdcTablePolicy = service:newCdcTablePolicy("ItemQuantity").
tbl = service:GetTable("Item","PUB").
ASSIGN
    myCdcTablePolicy:Table            = tbl
    myCdcTablePolicy:Description      = "Records item quantity changes"
    myCdcTablePolicy:DataArea         = service:GetArea("customer index area")
    myCdcTablePolicy:IndexArea        = service:GetArea("customer index area")
    myCdcTablePolicy:Level            = CdcTablePolicyLevelEnum:Medium
    myCdcTablePolicy:IdentifyingField = YES
    myCdcTablePolicy:ChangeTableOwner = "PUB"
    myCdcTablePolicy:EncryptPolicy    = NO
    myCdcTablePolicy:State            = CdcTablePolicyStateEnum:inActive.
.

/* identifying field ItemNum */
myCdcFieldPolicy = service:NewCdcFieldPolicy().
myCdcTablePolicy:FieldPolicies:Add(myCdcFieldpolicy).
myCdcFieldPolicy:Field = tbl:Fields:Find("Item-Num").
myCdcFieldPolicy:IdentifyingField = 1.

/* Track OnHand field */
myCdcFieldPolicy = service:NewCdcFieldPolicy().
myCdcTablePolicy:FieldPolicies:Add(myCdcFieldpolicy).
myCdcFieldPolicy:Field = tbl:Fields:Find("On-Hand").
/* add policy to the collection */
tablePolicySet:Add(myCdcTablePolicy).


/* create a CDC policy for the orderline table */
myCdcTablePolicy = service:newCdcTablePolicy("OrderlineQuantity").
tbl = service:GetTable("Order-Line","PUB").

ASSIGN
    myCdcTablePolicy:Table            = tbl
    myCdcTablePolicy:Description      = "Records ordered items"
    myCdcTablePolicy:DataArea         = service:GetArea("customer index area")
    myCdcTablePolicy:IndexArea        = service:GetArea("customer index area")
    myCdcTablePolicy:Level            = CdcTablePolicyLevelEnum:Medium
    myCdcTablePolicy:IdentifyingField = YES
    myCdcTablePolicy:ChangeTableOwner = "PUB"
    myCdcTablePolicy:EncryptPolicy    = NO
    myCdcTablePolicy:State            = CdcTablePolicyStateEnum:inActive.
.

/* identifying field ItemNum */
myCdcFieldPolicy = service:NewCdcFieldPolicy().
myCdcTablePolicy:FieldPolicies:Add(myCdcFieldpolicy).
myCdcFieldPolicy:Field = tbl:Fields:Find("Item-Num").
myCdcFieldPolicy:IdentifyingField = 1.

/* Track Qty field */
myCdcFieldPolicy = service:NewCdcFieldPolicy().
myCdcTablePolicy:FieldPolicies:Add(myCdcFieldpolicy).
myCdcFieldPolicy:Field = tbl:Fields:Find("Qty").
```

```
tablePolicySet:Add(myCdcTablePolicy).

/* This create method creates the collection of policies */
service:CreateCdcTablePolicies(tablePolicyset).
```

Get CDC policies

The following code is an example for listing CDC table policies:

```
using OpenEdge.DataAdmin.Error.DataAdminErrorHandler from propath.
using OpenEdge.DataAdmin.ICdcTablePolicy from propath.
using OpenEdge.DataAdmin.ICdcTablePolicyset from propath.
using OpenEdge.DataAdmin.ICdcFieldPolicySet from propath.
using OpenEdge.DataAdmin.ICdcFieldPolicy from propath.
using OpenEdge.DataAdmin.Lang.Collections.IIterator from propath.

define variable service       as OpenEdge.DataAdmin.DataAdminService no-undo.
define variable errorHandler  as DataAdminErrorHandler               no-undo.
define variable tablePolicy   as ICdcTablePolicy                     no-undo.
define variable tablePolicies as ICdcTablePolicyset                  no-undo.
define variable FieldPolicies as ICdcFieldPolicyset                  no-undo.
define variable FieldPolicy   as ICdcFieldPolicy                     no-undo.
define variable iter          AS IIterator                            no-undo.

service = new OpenEdge.DataAdmin.DataAdminService(ldbname(1)).

/* get policy based on policy name */
tablePolicy = service:GetCdcTablePolicy("Invoiceactive").
FieldPolicies = tablePolicy:FieldPolicies.
fieldpolicy = FieldPolicies:Find("id").

/* get all policies */
tablePolicies = service:GetCdcTablePolicies().


/* get policy set based on filter */
tablePolicies = service:GetCdcTablePolicies("TableName= ~"Customer~"").

iter = tablePolicies:Iterator().

do while iter:HasNext():
    tablepolicy = cast(iter:Next(), ICdcTablePolicy).

    message tablePolicy:Name view-as alert-box.

end.
```

Update CDC policies

The following code is an example for updating CDC table policies:

```
using OpenEdge.DataAdmin.Error.DataAdminErrorHandler from propath.
using OpenEdge.DataAdmin.ICdcTablePolicy from propath.
using OpenEdge.DataAdmin.ICdcTablePolicyset from propath.
using OpenEdge.DataAdmin.ICdcFieldPolicySet from propath.
using OpenEdge.DataAdmin.ICdcFieldPolicy from propath.
using OpenEdge.DataAdmin.ITable from propath.
using OpenEdge.DataAdmin.cdctablepolicystateenum.
using OpenEdge.DataAdmin.cdctablepolicylevelenum.

define variable service as OpenEdge.DataAdmin.DataAdminService no-undo.
define variable errorHandler as DataAdminErrorHandler no-undo.
define variable tablePolicy as ICdcTablePolicy no-undo.
define variable tablePolicies as ICdcTablePolicyset no-undo.
define variable FieldPolicies as ICdcFieldPolicyset no-undo.
define variable FieldPolicy as ICdcFieldPolicy no-undo.
define variable tbl as ITable no-undo.

service = new OpenEdge.DataAdmin.DataAdminService(ldbname(1)).

tablePolicy = service:GetCdcTablePolicy("MyPolicy").
FieldPolicies = tablePolicy:FieldPolicies.
tbl = service:GetTable(tablePolicy:Name).

/* update table policy properties */
tablepolicy:DataArea = service:Getarea("Primary Index Area").
tablepolicy:level = cdctablepolicylevelenum:Maximum.
tablepolicy:state = cdctablepolicystateenum:active.
tablepolicy:Name = "nameChanged".
tablepolicy:EncryptPolicy = yes.
tablepolicy:Description = "modified".
tablepolicy:IdentifyingField = no.

/* to add new field policy */
fieldPolicy = service:NewCdcFieldPolicy().
tablePolicy:FieldPolicies:add(fieldPolicy).
fieldPolicy:Field = tbl:Fields:Find("Amount").
fieldPolicy:IdentifyingField = 3.

/* modify existing field policy */
fieldPolicy = fieldpolicies:Find("Country").
fieldPolicy:IdentifyingField = 3.

service:UpdateCdcTablePolicy(tablePolicy).
```

Delete CDC policies

The following code is an example for deleting CDC table policies:

```
using OpenEdge.DataAdmin.Error.DataAdminErrorHandler from propath.
using OpenEdge.DataAdmin.ICdcTablePolicy from propath.
using OpenEdge.DataAdmin.ICdcTablePolicyset from propath.

define variable service      as OpenEdge.DataAdmin.DataAdminService          no-undo.
define variable errorHandler  as DataAdminErrorHandler                       no-undo.
define variable tablePolicies as ICdcTablePolicyset                          no-undo.
define variable tablepolicy   as ICdcTablePolicy                             no-undo.
define variable iter         as OpenEdge.DataAdmin.Lang.Collections.IIterator no-undo.

service = new OpenEdge.DataAdmin.DataAdminService(ldbname(1)).

/* delete policy with policy name */
service:DeleteCdcTablePolicy("policyname").

/* get all policies to delete */
tablePolicies = service:GetCdcTablePolicies().

/* iterate between policies to delete them */
iter = tablePolicies:Iterator().
do while iter:HasNext():
    tablepolicy = cast(iter:Next(), ICdcTablePolicy).
    service:DeleteCdcTablePolicy(tablepolicy:Name).
end.
```

Enums

Enum properties are used when table policies are created and these properties reside in the `OpenEdge.DataAdmin` package.

A list of enums defined for Change Data Capture are provided below:

```
ENUM CdcTablePolicyStateEnum:
 Active = 0
 Inactive = 1.
```

```
ENUM CdcTablePolicyLevelEnum:
 Minimal = 0
 MinimalWithBitMap = 1
 Medium = 2
 Maximum = 3.
```

Interface properties (Table policy)

The following table provides a list of properties for ICdcTablePolicy interface:

Table 79: ICdcTablePolicy properties

Property	Data type	Access	Description
Name	CHARACTER	Readable, Writeable	The name of the policy provided by the user.
CdcPolicyId	CHARACTER	Readable	The UUID generated by ABL Virtual Machine (AVM). This is a case-sensitive system field.
Description	CHARACTER	Readable, Writeable	(Optional) The description of the policy provided by the user.
State	CdcTablePolicyStateEnum	Readable, Writeable	The internal state of the policy. Set the value to CDCTablePolicyStateEnum: Active or CDCTablePolicyStateEnum: Inactive to make the policy active or inactive respectively.
Instance	CdcTablePolicyInstanceEnum	Readable	The policy on the source table. This is a system field. The instance values are 0, 1, and 2 for Current policy, Previous policy, and Pending policy respectively.
Table	ITable	Readable, Writeable	The source table from which the changes are captured. The source table can be a PUB or non-PUB table.
ChangeTable	CHARACTER	Readable, Writeable	The name of the change table created by AVM for this policy. This is a system field. **Note:** The value will be unknown until the policy is made active and the _Level column is greater than 0.
ChangeTableOwner	CHARACTER	Readable, Writeable	(Optional) The name of the non-PUB or PUB owner of the change table. If the non-PUB owner value is not set, the property chooses the PUB owner value as the default value.

Property	Data type	Access	Description
IdentifyingField	LOGICAL	Readable, Writeable	Set the value to YES to create an index on the change table for the fields in the related _Cdc-Field-Policy records that have a non-zero value in the _Identifying-Field column of the _Cdc-Field-Policy.
Level	CdcTablePolicyLevelEnum	Readable, Writeable	The level of data that is captured on the source table. The levels are: • Minimal(0) • MinimalWithBitMap(1) • Medium(2) • Maximum(3)
EncryptPolicy	LOGICAL	Readable, Writeable	The encryption policy used for the change table. When the source table is encrypted and the value is set to YES, the AVM uses the same encryption policy of the source table to encrypt the change table.
DataArea	IArea	Readable, Writeable	The storage area where the change table is created by AVM. This is a Type II area. The AVM creates a change table when the value of _Level column is 1, 2 or 3.
IndexArea	IArea	Readable, Writeable	The storage area where the change table indices are created by AVM. This is a Type II area. The AVM creates change table indices when the value of _Level column is 1, 2 or 3 and the _Identifying-Field is set to YES.
FieldPolicies	ICdcFieldPolicySet	Readable, Writeable	The collection of CDC field policies of a CDC table policy.
LastModified	DateTime-TZ	Readable	The time the _Policy-Instance was last modified.

Interface properties (Field policy)

The following table provides a list of properties for `ICdcFieldPolicy` interface:

Table 80: ICdcFieldPolicy properties

Property	Data type	Access	Description
`Field`	`IField`	Readable, Writeable	The selected field of the source table whose values are tracked for changes. **Note:** The identified column cannot be a Large Object (LOB) column.
`IdentifyingField`	`INTEGER`	Readable, Writeable	Specifies whether the column must be indexed in the change table. An unknown value indicates that the column will not be indexed. A non-null value (greater than 0) indicates the ordinal position of the column in the index. **Note:** To set a non-null value for this column, ensure that the `_Cdc-Table-Policy._Identifying-Fields` is set to YES in the parent `_Cdc-Table-Policy`.
`CdcPolicyId`	`CHARACTER`	Readable	The `_Policy-Id` of the parent `_Cdc-Table-Policy` row.
`FieldPosition`	`INTEGER`	Readable	The physical position number of the column in the source file whose values are tracked for changes. This is a system field.
`FieldRecId`	`INTEGER`	Readable	RecId of the `_Field` row for the column from the source table whole values are tracked for changes.

API to generate Change Data Capture policy program

The following sample code shows how to use API service to generate an ABL program for a CDC policy:

```
using OpenEdge.DataAdmin.ICdcTablePolicy from propath.

define variable service as OpenEdge.DataAdmin.DataAdminService no-undo.
define variable tablePolicy as ICdcTablePolicy no-undo.

service = new OpenEdge.DataAdmin.DataAdminService(ldbname(1)).
tablePolicy = service:GetCdcTablePolicy("customerinActive2").
tablePolicy:ExportAsProcedure("cdcpoliy.p").
```

The `ExportAsProcedure( )` method takes the file path, where the policy is to be generated, as input parameter. Note that the CDC table policy must exist to generate the policy program. In this case, the CDC table policy is `customerinActive2`.

ABL procedure to dump

This ABL procedure dumps Change Data Capture (CDC) policies from your database.

Note: Before you begin, ensure that the database is running and Change Data Capture is enabled.

Run the following procedure from your Procedure Editor window:

Syntax

```
run prodict/dump_cdc.p( policy-name,output-dir,codepage ).
```

Parameters

policy-name

Specify the policy name(s) to dump. All valid values must be surrounded by quotations and they include:

- "ALL" — Dump all current policies for the database.
- *"policy-name"* — Dump the named policy.
- *"policy-list"* — Dump the policy list. The list must be a comma-separated list.

Note: If you specify a non-existing policy to dump, the dumping procedure reports an error.

output-dir

Specify the directory where the `.cd` file that has the dump data of the specified policies is to be created. This directory is relative to your defined working directory.

codepage

Specify the code page format in which you want the CDC policies to be dumped. Valid values include:

- ? — No code page conversions occur.
- "" — Default code page conversions occur.
- *"target-codepage"* — The code page to which the target object is converted.

Example

You can also run the dump procedure from the **proenv** command line. The following example demonstrates dumping all the policies from a database named `cdc_sample_db`, to the working directory with the default code page.

```
proenv> prowin -db cdc_sample_db -p dump.p -b
```

dump.p

```
run prodict/dump_cdc.p( "all",".","" ).
```

Understanding the structure of a .cd file

When you open the `.cd` file, the following contents appear:

- **First row** — Contains information of a single CDC table policy. This row has an identifier or tag called “Table” as a prefix to identify that the policy is a CDC table policy. The value at the end of the row indicates the number of field policies associated with the CDC table policy.

  Each column in the first row appears in the following oder:

Column	Description
`Table`	The identifier or tag to identify that the policy is a CDC table policy.
`Policy Name`	The name of the table policy.
`Description`	The description of the policy.
`State`	The internal state of the policy. The values are 0 and 1 for Active and Inactive respectively.
`Instance`	The policy on the source table. The values are 0, 1, and 2 for Current policy, Previous policy, and Pending policy respectively.
`Source Table`	The name of the source table.
`Source Table Owner`	The name of the owner of the source table. The source table can be a PUB or non-PUB table.
`Data Area`	The storage area where the change table.
`Index Area`	The storage area where the change table indices are created.
`Identifying Fields`	Indicates if the Identifying fields are available for the policy. The values are Yes and No.
`Level`	The level of data that is captured on the source table. The levels are: • Minimal(0) • MinimalWithBitMap(1) • Medium(2) • Maximum(3)
`Change Table`	The name of the change table.
`Change Table Owner`	The name of the non-PUB or PUB owner of the change table.
`Encrypt Policy`	The encryption policy used for the change table. The values are Yes and No.
`Misc Fields`	This is an extent [16] field and the values are non-editable. All values of `_Misc` fields for `_Cdc-Table-Policy` are dumped.

Column	Description
Field Policy Count	The number of field policies available for a table policy. A value of zero (0) indicates that there are no field policies available for that table policy.

For a CDC table policy in a .cd file, the following fields are ignored:

- _Policy-Id
- _Last-Modified
- _ObjectId
- _First-User-Field

Though the _Policy-Instance field is dumped, it can be overridden when loading the .cd file.

Note: You cannot set the _Policy-Instance field value directly. To set the required _Policy-Instance field value, you must change the _Policy-State field value during the loading process.

In the _Cdc-Table-Policy table, the name of the source table is identified from the _Source-File-Recid, and the data and index areas are identified from the _Area-Ianum and _Index-Area-Ianum fields.

- **Subsequent rows** — Contain information of the field policies associated with the CDC table policy. These rows have an identifier or tag called “Field” as a prefix to identify that the policies are field policies, followed by the name of the CDC policy name that they belong to.

  Each column in these rows appears in the following order:

Column	Description
`Field`	The identifier or tag to identify that the policy is a CDC field policy.
`Policy Name`	The name of the corresponding CDC table policy.
`Field Name`	The name of the field.
`Identifying Fields`	Indicates if the Identifying fields are available for the policy. The values range from 1 to 15 and indicate ordinal position in the index.
`Misc Fields`	This is an extent [16] field and the values are non-editable. All values of `_Misc` fields for `_Cdc-Field-Policy` are dumped.

  For a CDC field policy in a `.cd` file, the following fields are ignored:

  - `_Policy-Id`
  - `_Field-Position`
  - `_Field-Recid`

  In the `_Cdc-Field-Policy` table, the name of the field is identified from the `_Field-Recid` field.

- **Trailer** — Contains information of the `.cd` file such as its name, the number of CDC table policies available in the file, the name of the database from which the policies are dumped, time and date when the dumping process completed, and format of the code page.

  Each row in the trailer appears in the following order:

Row	Description
`filename`	The name of the `.cd` dump file.
`records`	The number of CDC table policies available in the dump file.
`ldbname`	The name of the database from which the policies are dumped.
`timestamp`	The time and date when the policies are dumped.
`numformat`	The numeric format
`dateformat`	The format of the date when the policies are dumped.
`cpstream`	The format of the code page in which you want the CDC policies to be dumped.

The following example displays the contents of a sample .cd file:

```
"Table" "cust" ? 0 0 "Customer" "PUB" "Order Index Area" "Order Index Area" no 0 ? "PUB"
 no ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? 0
"Table" "pol" ? 0 0 "Order" "PUB" "Order Index Area" "Order Index Area" no 1 "myCT"
"PUB" no ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? 2
"Field" "pol" "Carrier" ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
"Field" "pol" "Cust-Num" ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ? ?
.
PSC
filename=sptest
records=0000000000002
ldbname=sptest
timestamp=2016/11/24-21:06:42
numformat=44,46
dateformat=mdy-1950
cpstream=ISO8859-1
.
0000000377
```

ABL procedure to load

This ABL procedure loads CDC policies to your database.

Note: Before you begin, ensure that the database is running and Change Data Capture is enabled.

Run the following procedure from your Procedure Editor window:

Syntax

```
run prodict/load_cdc.p( file-name,input-dir ).
```

Parameters

file-name

Specify the file name without .cd extension. A valid value must be surrounded by quotations:

- *"file-name"* — Load the file which has CDC policy information.

input-dir

Specify the directory from where the .cd file is to be loaded. This directory is relative to your defined working directory.

Example

You can also run the load procedure from the **proenv** command line. The following example demonstrates loading all the policies to a database named cdc_sample_db, from the working directory.

```
proenv> prowin -db cdc_sample_db -p load.p -b
```

load.p

```
run prodict/load_cdc.p( "OrderPolicy","c:\load" ).
```

ABL support classes for CDC

The following class an enumumeration support ABL developers writing ETL programs.

CDCTrackingHelper class

The `CDCTrackingHelper` class provides convenience methods for accessing data in a `_Cdc-Change-Tracking` record and its related change table record(s). It is specific to a given CDC source table. An ETL program can create an instance of `CDCTrackingHelper` for the source table being processed. The ETL program creates a `CDCTrackingHelper` object for a `_Cdc-Change-Tracking` buffer, then iterates through the `_Cdc-Change-Tracking` records using this buffer. The ETL program can then use the convenience methods in the `CDCTrackingHelper` object to access the record currently in the buffer.

OpenEdge.DataAdmin.Util.CDCTrackingHelper public methods

The following public methods are defined by the interface:

- `CONSTRUCTOR CDCTrackingHelper( INPUT SourceTableName AS CHARACTER, INPUT ChangeTrackingHandle AS HANDLE)`
- `CONSTRUCTOR CDCTrackingHelper( INPUT SourceTableName AS CHARACTER, INPUT TableOwner AS CHARACTER, INPUT ChangeTrackingHandle AS HANDLE)`

OpenEdge.DataAdmin.Util.CDCTrackingHelper properties

The following properties are defined by the interface:

- `SourceTableNumber`
- `GetChangedFields()`
- `FieldChanged(INPUT fieldName as CHARACTER)`
- `GetOperation()`
- `IsUpdate()`
- `IsCreate()`
- `IsDelete()`
- `IsRecordSplit()`
- `IsRecordSplit(INPUT before AS LOGICAL)`
- `FetchChangeRecord(INPUT hbuf AS HANDLE)`
- `FetchChangeRecord(INPUT hbuf AS HANDLE, INPUT before AS LOGICAL)`
- `CreateChangeBuffer()`
- `DeleteChangeTrackingRecord()`

CDCOperation enumeration

The `OpenEdge.DataAdmin.Util.CDCOperation` enumumeration provides an enumeration member for each of the possible CDC `_Operation` values.

Members

The following table lists the members of the enumeration.

`Create`	`Delete`	`AfterUpdate`
`BeforeUpdate`	`FieldDelete`	`FieldAdd`
`PartitionDrop`	`PartitionTruncate`	`PartitionDeallocate`
`MTPartitionDeallocate`	`MTPartitionDrop`	`MergePartitionDrop`

ABL examples

The following code samples demonstrate the use of the CDCTrackingHelper class in an ETL programs.

Parsing potentially split change table rows for the Order change table

```
DEF VAR hbuf   AS HANDLE NO-UNDO.
DEF VAR hbuf2  AS HANDLE NO-UNDO.
DEF VAR ohelper AS CDCTrackingHelper NO-UNDO.

/* Get a CDCTrackingHelper for the Order table, using the default buffer
 * of the _Cdc-Change-Tracking table */
ohelper = NEW CDCTrackingHelper("Order", BUFFER _Cdc-Change-Tracking:Handle).

/* get a buffer to help process any split records */
hbuf = ohelper:CreateChangeBuffer().
/* create a new buffer for the BEFORE change data */
CREATE BUFFER hbuf2 FOR TABLE hbuf.

/* Iterate through the change table rows */
FOR EACH _Cdc-Change-Tracking WHERE
    _Cdc-Change-Tracking._Source-Table-Number = ohelper:SourceTableNumber
    /* Rows are sorted in _Change-Sequence order */ :

    /* Is this an update row? */
    IF (ohelper:IsUpdate()) THEN
    DO:
        /* We only want to process if the order total changed */
        IF (ohelper:FieldChanged("OrderTotal")) THEN
        DO:
            /* Get the BEFORE change table row */
            ohelper:FetchChangeRecord(hbuf2, TRUE).
            /* Get the AFTER change table row.
             * We can use the other overload, as the _Operation in the
             * _Cdc-Change-Tracking and change table rows is the same for
             * AFTER  */
            ohelper:FetchChangeRecord(hbuf).
            /* perform whatever logic needs to occur for the ETL */
        END.
    END.

    /* purge rows in the _Cdc-Change-Tracking and change table,
     * we have finished with them */
    ohelper:DeleteChangeTrackingRecord().
END.
/* delete objects we created */
DELETE OBJECT hbuf2.
/* ohelper can be garbage collected */
```

Parsing change table row for the Order table, when it is known they do not split.

```
DEF BUFFER buf2 FOR CDC_Order.
DEF VAR oCDCUtil AS CDCUtility NO-UNDO.
DEF VAR ohelper AS CDCTrackingHelper NO-UNDO.

/* Get a CDCTrackingHelper for the Order table, using the default buffer
 * of the _Cdc-Change-Tracking table */
ohelper = NEW CDCTrackingHelper("Order", BUFFER _Cdc-Change-Tracking:Handle).

/* Iterate through the change table rows */
FOR EACH _Cdc-Change-Tracking WHERE
    _Cdc-Change-Tracking._Source-Table-Number = ohelper:SourceTableNumber
    /* Rows are sorted in _Change-Sequence order */ :

    IF (ohelper:IsUpdate()) THEN
    DO:
        /* We only want to process if the order total changed */
        IF (ohelper:FieldChanged("OrderTotal")) THEN
        DO:
            /* Get the BEFORE change table row.
             * We are not using the convenience FetchChangeRecord
             * method, so we need to use the integer _Operation values */
            FIND FIRST buf2 WHERE
                buf2._Change-Sequence = _Cdc-Change-Tracking._Change-Sequence AND
                buf2._Operation = INT(CDCOperation:BeforeUpdate) NO-ERROR.
            /* Get the AFTER change table row. */
            FIND FIRST buf WHERE
                buf._Change-Sequence = _Cdc-Change-Tracking._Change-Sequence AND
                buf._Operation = INT(CDCOperation:AfterUpdate) NO-ERROR.
            /* perform whatever logic needs to occur for the ETL */
        END.
    END.
    /* purge rows in the _Cdc-Change-Tracking and change table,
     * we have finished with them */
    ohelper:DeleteChangeTrackingRecord().
END.
/* ohelper can be garbage collected */
```

D

HLC Library Function Reference

HLC library functions provide an interface between your HLC functions and ABL (Advanced Business Language).

Note: The HLC interface is supported both by the single-threaded ABL client and the PAS for OpenEdge multi-session agent.

For more information on using these functions, see Host Language Call Interface on page 375

For details, see the following topics:

- Function summary
- Function reference

Function summary

All HLC library function names begin with the pro prefix. The tables that follow provide a brief description of each function, listed according to their application functionality.

Shared-variable access

The following table lists HLC library functions that access ABL shared variables. Each function operates on variables with a specific ABL data type.

Table 81: Functions that access ABL shared variables

Function	Description
prordc( )	Reads the value of a shared CHARACTER variable
prordd( )	Reads the value of a shared DATE variable
prordi( )	Reads the value of a shared INTEGER variable
prordl( )	Reads the value of a shared LOGICAL variable
prordn( )	Reads the value of a shared DECIMAL variable
prordr( )	Reads the value of a shared RECID variable
prowtc( )	Writes a value to a shared CHARACTER variable
prowtd( )	Writes a value to a shared DATE variable
prowti( )	Writes a value to a shared INTEGER variable
prowtl( )	Writes a value to a shared LOGICAL variable
prowtn( )	Writes a value to a shared DECIMAL variable
prowtr( )	Writes a value to a shared RECID variable

Shared-buffer access

The following table lists HLC library functions that access the fields of a shared buffer defined during an ABL transaction. Each function operates on fields with a specific ABL data type.

Table 82: Functions that access ABL shared buffers

Function	Description
profldix( )	Returns the handle to a specified field in a shared buffer
prordbc( )	Reads the value of a CHARACTER field in a shared buffer
prordbd( )	Reads the value of a DATE field in a shared buffer
prordbi( )	Reads the value of an INTEGER field in a shared buffer
prordbl( )	Reads the value of a LOGICAL field in a shared buffer
prordbn( )	Reads the value of a DECIMAL field in a shared buffer
prordbr( )	Reads the value of a RECID field in a shared buffer
prowtbc( )	Writes a value to a CHARACTER field in a shared buffer

Function	Description
prowtbd( )	Writes a value to a DATE field in a shared buffer
prowtbi( )	Writes a value to an INTEGER field in a shared buffer
prowtbl( )	Writes a value to a LOGICAL field in a shared buffer
prowtbn( )	Writes a value to a DECIMAL field in a shared buffer
prowtbr( )	Writes a value to a RECID field in a shared buffer

Screen display

The following table lists HLC library functions that enable HLC applications to use the screen display in a manner consistent with ABL display standards.

Table 83: Functions that interact with ABL displays

Function	Description
proscopn( )	Restores a UNIX terminal to its I/O mode before OpenEdge was run
proclear( )	Clears the display
prosccls( )	Sets a UNIX terminal to raw mode
promsgd( )	Displays a message on the display (in any I/O mode)

Interrupt handling

The following table lists HLC library functions that allow applications to test for specific ABL interrupts.

Table 84: Functions that test for ABL interrupts

Function	Description
prockint( )	Tests whether the interrupt key (**STOP** key) has been pressed

Timer-service routines

The following table lists HLC library functions that access ABL-managed timer services.

Table 85: Functions that access ABL timer services

Function	Description
prosleep( )	Suspends execution for a specified time interval

Function	Description
`proevt( )`	Sets a specified interval timer
`prowait( )`	Suspends execution until a specified interval timer (previously set by proevt( )) expires
`procncel( )`	Cancels a specified interval timer previously set by proevt( )

Function reference

Following is an alphabetical listing of HLC library functions, including calling sequences. Certain library functions accept input parameters that point to character strings containing the names of ABL shared variables (*psymnam*), buffers (*pbufnam*), and fields (*pfldnam*). These names are not case sensitive.

prockint( ) - Test for Interrupt Key

Tests whether the user pressed the interrupt key (the **STOP** key or an assigned function key equivalent).

If the user pressed the interrupt key, `prockint( )` returns 1; otherwise, `prockint( )` returns 0:

Syntax

```
int
prockint ( )
```

proclear( ) - Clear the Display

Clears the display.

On successful completion, `proclear( )` returns 0; otherwise, it returns a non-zero value:

Syntax

```
int
proclear ( )
```

Note:

- The `proclear( )` function applies only to character-mode systems.
- On character-mode systems, `proclear( )` works whether the terminal is in raw or cooked mode. For more information on raw- and cooked-mode terminal I/O, see Host Language Call Interface on page 375.

See Also:

promsgd( ) - Display Message on page 676, prosccls( ) - Set Terminal To Raw Mode and Refresh on page 688, proscopn( ) - Set Terminal To Initial Mode on page 689

procncel( ) - Cancel Interval Timer

Cancels a specified interval timer previously set by `proevt( )`. You can call this function even if the time interval has already expired:

Syntax

```
void
procncel ( pflag )
  char *pflag;
```

pflag

Points to a timer flag previously passed to `proevt( )` to start the specified time interval. This flag specifies one of many interval timers that you can set with `proevt( )`.

Note: This function applies only to UNIX systems. For more information on using interval timers, see Host Language Call Interface on page 375.

See Also:

prockint( ) - Test for Interrupt Key on page 674, proevt( ) - Set Interval Timer on page 675, prosleep( ) - Sleep For Specified Interval on page 689, prowait( ) - Wait For Timer To Expire on page 689

proevt( ) - Set Interval Timer

Sets an interval timer specified by a timer flag. The caller must test the timer flag to know whether the time interval has expired for the corresponding interval timer. The caller must not free the storage containing the flag until either the time interval has expired or `procncel( )` has been called to cancel the specified interval timer. You can use `proevt( )` to start an unlimited number of times:

Syntax

```
void
proevt ( seconds, pflag )
  int  seconds ;
  char *pflag;
```

seconds

Specifies the duration (in seconds) to set the specified interval timer.

pflag

Points to a timer flag used to identify and monitor the specified interval timer. The caller must set **pflag* to 0 before invoking `proevt( )`. When the time interval expires, `proevt( )` sets **pflag*

to a non-zero value. You can test *`*pflag`* explicitly with your own code or implicitly using the `prowait( )` function.

Note: This function applies only to UNIX systems. For more information on using HLC timer services, see Host Language Call Interface on page 375.

See Also:

procncel( ) - Cancel Interval Timer on page 675, prockint( ) - Test for Interrupt Key on page 674, prosleep( ) - Sleep For Specified Interval on page 689, prowait( ) - Wait For Timer To Expire on page 689

profldix( ) - Return Field Handle

Returns a field handle for a specified field within a shared buffer. The handle is returned as an integer value. The shared buffer can also refer to a `TEMP-TABLE`.

On successful completion, `profldix( )` returns the field handle for the field; otherwise, it returns a negative value:

Syntax

```
int
profldix ( pbufnam, pfldnam )
  char *pbufnam;
  char *pfldnam;
```

`pbufnam`

Points to the name of the shared buffer containing the specified field. You supply the name from your OpenEdge application.

`pfldnam`

Points to the name of the specified field within the buffer. You supply the name from your OpenEdge application.

Note: All other shared buffer access functions require the field handle that `profldix( )` returns.

See Also:

prordbc( ) - Read CHARACTER Field on page 677, prordbd( ) - Read DATE Field on page 678, prordbi( ) - Read INTEGER Field on page 679, prordbl( ) - Read LOGICAL Field on page 680, prordbn( ) - Read DECIMAL Field on page 681, prordbr( ) - Read RECID Field on page 682, prowtbc( ) - Write CHARACTER Field on page 690, prowtbd( ) - Write DATE Field on page 691, prowtbi( ) - Write INTEGER Field on page 692, prowtbl( ) - Write LOGICAL Field on page 693, prowtbn( ) - Write DECIMAL Field on page 693, prowtbr( ) - Write RECID Field on page 694

promsgd( ) - Display Message

The `promsgd( )` function displays messages on the display.

On successful completion, `promsgd( )` returns a 0; otherwise, it returns a non-zero value:

Syntax

```
int
promsgd ( pmessage )
  char *pmessage;
```

pmessage

This input parameter points to the message you want to display.

Note:

- On UNIX character-mode systems, when the terminal is under the control of ABL (in raw-mode terminal I/O), the messages appear at the bottom of the display. If the application previously set the terminal to cooked mode with a call to `proscopn( )`, the messages are written to the current output destination. If the current output destination is the display, the messages appear on the display in normal video, similar to the output of the standard C library function `printf( )`. For more information on raw- and cooked-mode terminal I/O in character mode, see Host Language Call Interface on page 375.
- In Windows, `promsgd( )` displays messages in an alert box. Raw and cooked terminal I/O apply only to OpenEdge running on UNIX character-mode systems.

See Also:

proclear( ) - Clear the Display on page 674, prosccls( ) - Set Terminal To Raw Mode and Refresh on page 688, proscopn( ) - Set Terminal To Initial Mode on page 689

prordbc( ) - Read CHARACTER Field

The `prordbc( )` function reads the value of a CHARACTER field in a shared buffer.

On successful completion, `prordbc( )` returns 0; otherwise, it returns a non-zero value:

Syntax

```
int
prordbc ( pbufnam, fhandle, index, pvar, punknown, varlen,
          pactlen )
  char *pbufnam;
  int   fhandle;
  int   index;
  char *pvar;
  int  *punknown;
  int   varlen;
  int  *pactlen;
```

pbufnam

This input parameter points to the name of the specified shared buffer. You supply the name from your OpenEdge application.

fhandle

This input parameter is the field handle that `profldix( )` returns for the specified field.

index

This input parameter specifies an index value for an array field. If the field is not an array, you must set the value of *index* to 0.

pvar

This output parameter points to a buffer where `prordbc( )` returns the value of the specified CHARACTER field.

punknown

This output parameter points to an integer where `prordbc( )` returns 1 if the field has the Unknown value (?), and returns 0 otherwise.

varlen

This input parameter contains the length of the buffer that *pvar* specifies.

pactlen

This output parameter points to an integer where `prordbc( )` returns the actual length (in bytes) of the data in the specified CHARACTER field.

See Also:

profldix( ) - Return Field Handle on page 676, prordbd( ) - Read DATE Field on page 678, prordbi( ) - Read INTEGER Field on page 679, prordbl( ) - Read LOGICAL Field on page 680, prordbn( ) - Read DECIMAL Field on page 681, prordbr( ) - Read RECID Field on page 682, prowtbc( ) - Write CHARACTER Field on page 690

prordbd( ) - Read DATE Field

The `prordbd( )` function reads the value of a DATE field in a shared buffer.

On successful completion, `prordbd( )` returns 0; otherwise, it returns a non-zero value:

Syntax

```
int
prordbd ( pbufnam, fhandle, index, pyear, pmonth, pday,
          punknown )
  char *pbufnam;
  int   fhandle;
  int   index;
  int  *pyear;
  int  *pmonth;
  int  *pday;
  int  *punknown;
```

pbufnam

This input parameter points to the name of the specified shared buffer.

fhandle

This input parameter is the field handle that `profldix( )` returns for the specified field.

index

This input parameter specifies an index value for an array field. If the field is not an array, you must set the value of *index* to 0.

pyear

This output parameter points to an integer where `prordbd( )` returns the value of year.

pmonth

This output parameter points to an integer where `prordbd( )` returns the value of month.

pday

This output parameter points to an integer where `prordbd( )` returns the value of day.

punknown

This output parameter points to an integer where `prordbd( )` returns 1 if the field has Unknown value (?), and returns 0 otherwise.

See Also:

profldix( ) - Return Field Handle on page 676, prordbc( ) - Read CHARACTER Field on page 677, prordbi( ) - Read INTEGER Field on page 679, prordbl( ) - Read LOGICAL Field on page 680, prordbn( ) - Read DECIMAL Field on page 681, prordbr( ) - Read RECID Field on page 682, prowtbd( ) - Write DATE Field on page 691

prordbi( ) - Read INTEGER Field

The `prordbi( )` function reads the value of an `INTEGER` field in a shared buffer.

On successful completion, `prordbi( )` returns 0; otherwise, it returns a non-zero value:

Syntax

```
int
prordbi ( pbufnam, fhandle, index, pvar, punknown )
  char *pbufnam;
  int   fhandle;
  int   index;
  long *pvar;
  int  *punknown;
```

pbufnam

This input parameter points to the name of the specified shared buffer. You supply the name from your OpenEdge application.

fhandle

This input parameter is the field handle that `profldix( )` returns for the specified field.

index

This input parameter specifies an index value for an array field. If the field is not an array, you must set the value of *index* to 0.

pvar

This output parameter points to a long where `prordbi( )` returns the value of the specified `INTEGER` field.

punknown

This output parameter points to an integer where `prordbi( )` returns 1 if the field has the Unknown value (?), and returns 0 otherwise.

See Also:

profldix( ) - Return Field Handle on page 676, prordbc( ) - Read CHARACTER Field on page 677, prordbd( ) - Read DATE Field on page 678, prordbl( ) - Read LOGICAL Field on page 680, prordbn( ) - Read DECIMAL Field on page 681, prordbr( ) - Read RECID Field on page 682, prowtbi( ) - Write INTEGER Field on page 692

prordbl( ) - Read LOGICAL Field

The `prordbl( )` function reads the value of a `LOGICAL` field in a shared buffer.

On successful completion, `prordbl( )` returns 0; otherwise, it returns a non-zero value:

Syntax

```
int
prordbl ( pbufnam, fhandle, index, pvar, punknown )
  char *pbufnam;
  int   fhandle;
  int   index;
  int  *pvar;
  int  *punknown;
```

pbufnam

This input parameter points to the name of the specified shared buffer. You supply the name from your OpenEdge application.

fhandle

This input parameter is the field handle that `profldix( )` returns for the specified field.

index

This input parameter specifies an index value for an array field. If the field is not an array, you must set the value of *index* to 0.

pvar

This output parameter points to an integer where `prordbl( )` returns the value of the specified `LOGICAL` field. The `prordl( )` function returns 1 if the `LOGICAL` field is `TRUE`, and 0 otherwise.

-tielpunknown

This output parameter points to an integer where `prordbl( )` returns 1 if the field has the Unknown value (?), and returns 0 otherwise.

See Also:

profldix( ) - Return Field Handle on page 676, prordbc( ) - Read CHARACTER Field on page 677, prordbd( ) - Read DATE Field on page 678, prordbi( ) - Read INTEGER Field on page 679, prordbn( ) - Read DECIMAL Field on page 681, prordbr( ) - Read RECID Field on page 682, prowtbl( ) - Write LOGICAL Field on page 693

prordbn( ) - Read DECIMAL Field

The `prordbn( )` function reads the value of a `DECIMAL` numeric field in a shared buffer.

On successful completion, `prordbn( )` returns 0; otherwise, it returns a non-zero value:

Syntax

```
int
prordbn ( pbufnam, fhandle, index, pvar, punknown, varlen,
          pactlen )
  char *pbufnam;
  int   fhandle;
  int   index;
  char *pvar;
  int  *punknown;
  int   varlen;
  int  *pactlen;
```

pbufnam

This input parameter points to the name of the specified shared buffer. You supply the name from your OpenEdge application.

fhandle

This input parameter is the field handle that `profldix( )` returns for the specified field.

index

This input parameter specifies an index value for an array field. If the field is not an array, you must set the value of index to 0.

pvar

This output parameter points to a buffer where `prordbn( )` returns the value of the specified `DECIMAL` numeric field, formatted as a character string.

punknown

This output parameter points to an integer where prordbn( ) returns 1 if the field has the Unknown value (?), and returns 0 otherwise.

varlen

This input parameter contains the length of the buffer that pvar specifies.

pactlen

This output parameter points to an integer where prordbn( ) returns the actual length (in bytes) of the data in the specified DECIMAL numeric field.

See Also:

profldix( ) - Return Field Handle on page 676, prordbc( ) - Read CHARACTER Field on page 677, prordbd( ) - Read DATE Field on page 678, prordbi( ) - Read INTEGER Field on page 679, prordbl( ) - Read LOGICAL Field on page 680, prordbr( ) - Read RECID Field on page 682, prowtbn( ) - Write DECIMAL Field on page 693

prordbr( ) - Read RECID Field

The prordbr( ) function reads the value of a RECID field in a shared buffer.

On successful completion, prordbr( ) returns 0; otherwise, it returns a non-zero value:

Syntax

```
int
prordbr ( pbufnam, fhandle, index, pvar, punknown )
  char *pbufnam;
  int   fhandle;
  int   index;
  long *pvar;
  int  *punknown;
```

pbufnam

This input parameter points to the name of the specified shared buffer. You supply the name from your OpenEdge application.

fhandle

This input parameter is the field handle that profldix( ) returns for the specified field.

index

This input parameter specifies an index value for an array field. If the field is not an array, you must set the value of index to 0.

pvar

This output parameter points to a long where prordbr( ) returns the value of the specified RECID field.

punknown

This output parameter points to an integer where `prordbr( )` returns 1 if the field has the Unknown value (?), and returns 0 otherwise.

See Also:

profldix( ) - Return Field Handle on page 676, prordbc( ) - Read CHARACTER Field on page 677, prordbd( ) - Read DATE Field on page 678, prordbi( ) - Read INTEGER Field on page 679, prordbl( ) - Read LOGICAL Field on page 680, prordbn( ) - Read DECIMAL Field on page 681, prowtbr( ) - Write RECID Field on page 694

prordc( ) - Read CHARACTER Variable

The `prordc( )` function reads the value of a shared CHARACTER variable.

On successful completion, `prordc( )` returns 0; otherwise, it returns a non-zero value:

Syntax

```
int
prordc ( psymnam, index, pvar, punknown, varlen, pactlen )
  char *psymnam;
  int   index;
  char *pvar;
  int  *punknown;
  int   varlen;
  int  *pactlen;
```

psymnam

This input parameter points to the name of the specified shared CHARACTER variable. You supply the name from your OpenEdge application.

index

This input parameter specifies an index value for an array variable. If the variable is not an array, you must set the value of index to 0.

pvar

This output parameter points to a buffer where `prordc( )` returns the value of the specified CHARACTER variable.

punknown

This output parameter points to an integer where `prordc( )` returns 1 if the variable has the `Unknown value (?)`, and returns 0 otherwise.

varlen

This input parameter contains the length of the buffer that pvar specifies.

pactlen

This output parameter points to an integer where prordc( ) returns the actual length (in bytes) of the data in the specified CHARACTER variable.

See Also:

prordd( ) - Read DATE Variable on page 684, prordi( ) - Read INTEGER Variable on page 685, prordl( ) - Read LOGICAL Variable on page 685, prordn( ) - Read DECIMAL Variable on page 686, prordr( ) - Read RECID Variable on page 687, prowtc( ) - Write CHARACTER Variable on page 695

prordd( ) - Read DATE Variable

The prordd( ) function reads the value of a shared DATE variable.

On successful completion, prordd( ) returns 0; otherwise, it returns a non-zero value:

Syntax

```
int
prordd ( psymnam, index, pyear, pmonth, pday, punknown )
  char *psymnam;
  int   index;
  int  *pyear;
  int  *pmonth;
  int  *pday;
  int  *punknown;
```

psymnam

This input parameter points to the name of the specified shared DATE variable. You supply the name from your OpenEdge application.

index

This input parameter specifies an index value for an array variable. If the variable is not an array, you must set the value of index to 0.

pyear

This output parameter points to an integer where prordd( ) returns the value of year.

pmonth

This output parameter points to an integer where prordd( ) returns the value of month.

pday

This output parameter points to an integer where prordd( ) returns the value of day.

punknown

This output parameter points to an integer where prordd( ) returns 1 if the variable has the Unknown value (?), and returns 0 otherwise.

See Also:

prordc( ) - Read CHARACTER Variable on page 683, prordi( ) - Read INTEGER Variable on page 685, prordl( ) - Read LOGICAL Variable on page 685, prordn( ) - Read DECIMAL Variable on page 686, prordr( ) - Read RECID Variable on page 687, prowtd( ) - Write DATE Variable on page 696

prordi( ) - Read INTEGER Variable

The `prordi( )` function reads the value of a shared `INTEGER` variable.

On successful completion, `prodi( )` returns 0; otherwise, it returns a non-zero value:

Syntax

```
int
prordi ( psymnam, index, pvar, punknown )
  char *psymnam;
  int   index;
  long *pvar;
  int  *punknown;
```

psymnam

This input parameter points to the name of the specified shared `INTEGER` variable. You supply the name from your OpenEdge application.

index

This input parameter specifies an index value for an array variable. If the variable is not an array, you must set the value of index to 0.

pvar

This output parameter points to a buffer where `prordi( )` returns the value of the specified `INTEGER` variable.

punknown

This output parameter points to an integer where `prordi( )` returns 1 if the variable has the Unknown value (?), and returns 0 otherwise.

See Also:

prordc( ) - Read CHARACTER Variable on page 683, prordd( ) - Read DATE Variable on page 684, prordl( ) - Read LOGICAL Variable on page 685, prordn( ) - Read DECIMAL Variable on page 686, prordr( ) - Read RECID Variable on page 687, prowti( ) - Write INTEGER Variable on page 697

prordl( ) - Read LOGICAL Variable

The `prordl( )` function reads the value of a shared `LOGICAL` variable.

On successful completion, `prordl( )` returns 0; otherwise, it returns a non-zero value:

Syntax

```
int
prordl ( psymnam, index, pvar, punknown )
  char  *psymnam;
  int    index;
  int   *pvar;
  int   *punknown;
```

psymnam

This input parameter points to the name of the specified shared LOGICAL variable. You supply the name from your OpenEdge application.

index

This input parameter specifies an index value for an array variable. If the variable is not an array, you must set the value of index to 0.

pvar

This output parameter points to a buffer where prordl( ) returns the value of the specified LOGICAL variable. The prordbl( ) function returns 1 if the LOGICAL variable is TRUE, and 0 otherwise.

punknown

This output parameter points to an integer where prordl( ) returns 1 if the variable has the Unknown value (?), and returns 0 otherwise.

See Also:

prordc( ) - Read CHARACTER Variable on page 683, prordd( ) - Read DATE Variable on page 684, prordi( ) - Read INTEGER Variable on page 685, prordn( ) - Read DECIMAL Variable on page 686, prordr( ) - Read RECID Variable on page 687, prowtl( ) - Write LOGICAL Variable on page 698

prordn( ) - Read DECIMAL Variable

The prordn( ) function reads the value of a shared DECIMAL numeric variable.

On successful completion, prordn( ) returns 0; otherwise, it returns a non-zero value:

Syntax

```
int
prordn ( psymnam, index, pvar, punknown, varlen, pactlen )
  char *psymnam;
  int   index;
  char *pvar;
  int  *punknown;
  int   varlen;
  int  *pactlen;
```

psymnam

This input parameter points to the name of the specified shared DECIMAL numeric variable. You supply the name from your OpenEdge application.

index

This input parameter specifies an index value for an array variable. If the variable is not an array, you must set the value of index to 0.

pvar

This output parameter points to a buffer where prordn( ) returns the value of the specified DECIMAL numeric variable, formatted as a character string.

punknown

This output parameter points to an integer where prordn( ) returns 1 if the variable has the Unknown value (?), and returns 0 otherwise.

varlen

This input parameter contains the length of the buffer that pvar specifies.

pactlen

This output parameter points to an integer where prordn( ) returns the actual length of the character string (in bytes) to which the DECIMAL numeric variable is converted.

See Also:

prordc( ) - Read CHARACTER Variable on page 683, prordd( ) - Read DATE Variable on page 684, prordi( ) - Read INTEGER Variable on page 685, prordl( ) - Read LOGICAL Variable on page 685, prordr( ) - Read RECID Variable on page 687, prowtn( ) - Write DECIMAL Variable on page 699

prordr( ) - Read RECID Variable

The prordr( ) function reads the value of a shared RECID variable.

On successful completion, prordr( ) returns 0; otherwise, it returns a non-zero value:

Syntax

```
int
prordr ( psymnam, index, pvar, punknown )
  char *psymnam;
  int   index;
  long *pvar;
  int  *punknown;
```

psymnam

This input parameter points to the name of the specified shared RECID variable. You supply the name from your OpenEdge application.

index

This input parameter specifies an index value for an array variable. If the variable is not an array, you must set the value of index to 0.

pvar

This output parameter points to a long where `prordr( )` returns the value of the specified `RECID` variable.

punknown

This output parameter points to an integer where `prordr( )` returns 1 if the variable has the Unknown value (?), and returns 0 otherwise.

See Also:

prordc( ) - Read CHARACTER Variable on page 683, prordd( ) - Read DATE Variable on page 684, prordi( ) - Read INTEGER Variable on page 685, prordl( ) - Read LOGICAL Variable on page 685, prordn( ) - Read DECIMAL Variable on page 686, prowtr( ) - Write RECID Variable on page 699

prosccls( ) - Set Terminal To Raw Mode and Refresh

The `prosccls( )` function sets raw mode terminal I/O and refreshes the display. If you do not follow a call to `proscopn( )` with a call to `prosccls( )`, ABL still calls `prosccls( )` to refresh the frames on the display.

On successful completion, `prosccls( )` returns 0; if the display is not active (such as in batch mode), it returns 1:

Syntax

```
int
prosccls ( restore )
  int restore;
```

restore

If this input parameter has a non-zero value, the AVM refreshes the frames on the display. If it is 0, the AVM does not refresh the display.

Note: The `prosccls( )` function sets raw mode terminal I/O only on character-mode systems. On all other systems, `prosccls( )` just refreshes the display. For more information on terminal I/O modes, see Host Language Call Interface on page 375.

See Also:

proclear( ) - Clear the Display on page 674, promsgd( ) - Display Message on page 676, proscopn( ) - Set Terminal To Initial Mode on page 689

proscopn( ) - Set Terminal To Initial Mode

The `proscopn( )` function sets the terminal to the I/O mode that was in effect when OpenEdge started, which is usually cooked mode.

On successful completion, `proscopn( )` returns 0; otherwise, it returns a non-zero value:

Syntax

```
int
proscopn ( )
```

Note: The `proscopn( )` function applies only to character-mode systems. For more information on terminal I/O modes, see Host Language Call Interface on page 375.

See Also:

proclear( ) - Clear the Display on page 674, promsgd( ) - Display Message on page 676, prosccls( ) - Set Terminal To Raw Mode and Refresh on page 688

prosleep( ) - Sleep For Specified Interval

The `prosleep( )` function causes your application to suspend execution for a specified time interval. Use `prosleep( )` only in special circumstances; signals and interrupts do not interrupt this function:

Syntax

```
void
prosleep ( seconds )
  int seconds;
```

seconds

This input parameter specifies the number of seconds to suspend execution.

See Also:

prockint( ) - Test for Interrupt Key on page 674, procncel( ) - Cancel Interval Timer on page 675, proevt( ) - Set Interval Timer on page 675, prowait( ) - Wait For Timer To Expire on page 689

prowait( ) - Wait For Timer To Expire

The `prowait( )` function causes your application to suspend execution until a specified interval timer, set by the proevt( ) function, expires. Use `prowait( )` only in special circumstances; signals and interrupts do not interrupt this function:

Syntax

```
void
prowait ( pflag )
  char *pflag;
```

pflag

This input parameter points to a timer flag previously passed to `proevt( )` in order to start the specified time interval. This flag specifies one of many interval timers you can set with `proevt( )`.

Note: This function applies only to UNIX systems. For more information on using interval timers, see Host Language Call Interface on page 375.

See Also:

prockint( ) - Test for Interrupt Key on page 674, procncel( ) - Cancel Interval Timer on page 675, proevt( ) - Set Interval Timer on page 675, prosleep( ) - Sleep For Specified Interval on page 689

prowtbc( ) - Write CHARACTER Field

The `prowtbc( )` function writes a string value to a CHARACTER field in a shared buffer.

On successful completion, `prowtbc( )` returns 0; otherwise, it returns a non-zero value:

Syntax

```
int
prowtbc ( pbufnam, fhandle, index, pvar, unknown )
  char *pbufnam;
  int   fhandle;
  int   index;
  char *pvar;
  int   unknown;
```

pbufnam

This input parameter points to the name of the specified shared buffer. You supply the name from your OpenEdge application.

fhandle

This input parameter is the field handle that `profldix( )` returns for the specified field.

index

This input parameter specifies an index value for an array field. If the field is not an array, you must set the value of index to 0.

pvar

This input parameter points to a buffer containing the character string value for the specified CHARACTER field. The character string must be null-terminated.

unknown

If this input parameter contains the value 1, the Unknown value (?) is assigned to the variable. If this input parameter contains the value 0, the input value contained in pvar is assigned to the variable.

See Also:

profldix( ) - Return Field Handle on page 676, prordbc( ) - Read CHARACTER Field on page 677, prowtbd( ) - Write DATE Field on page 691, prowtbi( ) - Write INTEGER Field on page 692, prowtbl( ) - Write LOGICAL Field on page 693, prowtbn( ) - Write DECIMAL Field on page 693, prowtbr( ) - Write RECID Field on page 694

prowtbd( ) - Write DATE Field

The `prowtbd( )` function writes a date value to a `DATE` field in a shared buffer.

On successful completion, `prowtbd( )` returns 0; otherwise, it returns a non-zero value:

Syntax

```
int
prowtbd ( pbufnam, fhandle, index, year, month, day,
          unknown )
  char *pbufnam;
  int   fhandle;
  int   index;
  int   year;
  int   month;
  int   day;
  int   unknown;
```

`pbufnam`

This input parameter points to the name of the specified shared buffer. You supply the name from your OpenEdge application.

`fhandle`

This input parameter is the field handle that `profldix( )` returns for the specified field.

`index`

This input parameter specifies an index value for an array field. If the field is not an array, you must set the value of index to 0.

`year`

This input parameter contains the year value for the specified `DATE` field.

`month`

This input parameter contains the month value for the specified `DATE` field.

`day`

This input parameter contains the day value for the specified `DATE` field.

unknown

If this input parameter contains the value 1, the Unknown value (?) is assigned to the variable. If this input parameter contains the value 0, the input value contained in year, month, and day is assigned to the variable.

See Also:

profldix( ) - Return Field Handle on page 676, prordbd( ) - Read DATE Field on page 678, prowtbc( ) - Write CHARACTER Field on page 690, prowtbi( ) - Write INTEGER Field on page 692, prowtbl( ) - Write LOGICAL Field on page 693, prowtbn( ) - Write DECIMAL Field on page 693, prowtbr( ) - Write RECID Field on page 694

prowtbi( ) - Write INTEGER Field

The prowtbi( ) function writes an integer value to an INTEGER field in a shared buffer.

On successful completion, prowtbi( ) returns 0; otherwise, it returns a non-zero value:

Syntax

```
int
prowtbi ( pbufnam, fhandle, index, var, unknown )
  char *pbufnam;
  int   fhandle;
  int   index;
  long  var;
  int   unknown;
```

pbufnam

This input parameter points to the name of the specified shared buffer. You supply the name from your OpenEdge application.

fhandle

This input parameter is the field handle that profldix( ) returns for the specified field.

index

This input parameter specifies an index value for an array field. If the field is not an array, you must set the value of index to 0.

var

This input parameter contains the integer value for the specified INTEGER field.

unknown

If this input parameter contains the value 1, the Unknown value (?) is assigned to the variable. If this input parameter contains the value 0, the input value contained in var is assigned to the variable.

See Also:

profldix( ) - Return Field Handle on page 676, prordbi( ) - Read INTEGER Field on page 679, prowtbc( ) - Write CHARACTER Field on page 690, prowtbd( ) - Write DATE Field on page 691, prowtbl( ) - Write LOGICAL Field on page 693, prowtbn( ) - Write DECIMAL Field on page 693, prowtbr( ) - Write RECID Field on page 694

prowtbl( ) - Write LOGICAL Field

The `prowtbl( )` function writes a Boolean value to a `LOGICAL` field in a shared buffer.

On successful completion, `prowtbl( )` returns 0; otherwise, it returns a non-zero value:

Syntax

```
int
prowtbl ( pbufnam, fhandle, index, var, unknown )
  char *pbufnam;
  int   fhandle;
  int   index;
  long  var;
  int   unknown;
```

pbufnam

This input parameter points to the name of the specified shared buffer. You supply the name from your OpenEdge application.

fhandle

This input parameter is the field handle that `profldix( )` returns for the specified field.

index

This input parameter specifies an index value for an array field. If the field is not an array, you must set the value of index to 0.

var

This input parameter contains the boolean value for the specified `LOGICAL` field. Set var to 1 if you want to assign the value of the field to `TRUE`; set it to 0 otherwise.

unknown

If this input parameter contains the value 1, the Unknown value (?) is assigned to the variable. If this input parameter contains the value 0, the input value contained in var is assigned to the variable.

See Also:

profldix( ) - Return Field Handle on page 676, prordbl( ) - Read LOGICAL Field on page 680, prowtbc( ) - Write CHARACTER Field on page 690, prowtbd( ) - Write DATE Field on page 691, prowtbi( ) - Write INTEGER Field on page 692, prowtbn( ) - Write DECIMAL Field on page 693, prowtbr( ) - Write RECID Field on page 694

prowtbn( ) - Write DECIMAL Field

The `prowtbn( )` function writes a decimal value, formatted as a character string, to a `DECIMAL` numeric field in a shared buffer.

On successful completion, `prowtbn( )` returns 0; otherwise, it returns a non-zero value:

Syntax

```
int
prowtbn ( pbufnam, fhandle, index, pvar, unknown )
  char *pbufnam;
  int   fhandle;
  int   index;
  char *pvar;
  int   unknown;
```

pbufnam

This input parameter points to the name of the specified shared buffer. You supply the name from your OpenEdge application.

fhandle

This input parameter is the field handle that `profldix( )` returns for the specified field.

index

This input parameter specifies an index value for an array field. If the field is not an array, you must set the value of index to 0.

pvar

This input parameter points to a buffer containing the character string value for the specified DECIMAL numeric field. The character string must be null-terminated.

unknown

If this input parameter contains the value 1, the Unknown value (?) is assigned to the variable. If this input parameter contains the value 0, the input value contained in pvar is assigned to the variable.

See Also:

profldix( ) - Return Field Handle on page 676, prordbn( ) - Read DECIMAL Field on page 681, prowtbc( ) - Write CHARACTER Field on page 690, prowtbd( ) - Write DATE Field on page 691, prowtbi( ) - Write INTEGER Field on page 692, prowtbl( ) - Write LOGICAL Field on page 693, prowtbr( ) - Write RECID Field on page 694

prowtbr( ) - Write RECID Field

The `prowtbr( )` function writes an integer value to a `RECID` field in a shared buffer.

On successful completion, `prowtbr( )` returns 0; otherwise, it returns a non-zero value:

Syntax

```
int
prowtbr ( pbufnam, fhandle, index, var, unknown )
  char *pbufnam;
  int   fhandle;
  int   index;
  long  var;
  int   unknown;
```

pbufnam

This input parameter points to the name of the specified shared buffer. You supply the name from your OpenEdge application.

fhandle

This input parameter is the field handle that `profldix( )` returns for the specified field.

index

This input parameter specifies an index value for an array field. If the field is not an array, you must set the value of index to 0.

var

This input parameter contains the integer value for the specified `RECID` field.

unknown

If this input parameter contains the value 1, the Unknown value (?) is assigned to the variable. If this input parameter contains the value 0, the input value contained in var is assigned to the variable.

See Also:

profldix( ) - Return Field Handle on page 676, prordbr( ) - Read RECID Field on page 682, prowtbc( ) - Write CHARACTER Field on page 690, prowtbd( ) - Write DATE Field on page 691, prowtbi( ) - Write INTEGER Field on page 692, prowtbl( ) - Write LOGICAL Field on page 693, prowtbn( ) - Write DECIMAL Field on page 693

prowtc( ) - Write CHARACTER Variable

The `prowtc( )` function writes a character-string value to a shared `CHARACTER` variable.

On successful completion, `prowtc( )` returns 0; otherwise, it returns a non-zero value.

Syntax

```
int
prowtc ( psymnam, index, pvar, unknown )
  char *psymnam;
  int   index;
  char *pvar;
  int   unknown;
```

psymnam

This input parameter points to the name of the specified shared variable. You supply the name from your OpenEdge application.

index

This input parameter specifies an index value for an array variable. If the variable is not an array, you must set the value of index to 0.

pvar

This input parameter points to a buffer containing the character string value for the specified CHARACTER variable. The character string must be null-terminated.

unknown

If this input parameter contains the value 1, the Unknown value (?) is assigned to the variable. If this input parameter contains the value 0, the input value contained in pvar is assigned to the variable.

See Also:

prowtd( ) - Write DATE Variable on page 696, prowti( ) - Write INTEGER Variable on page 697, prowtl( ) - Write LOGICAL Variable on page 698, prowtn( ) - Write DECIMAL Variable on page 699, prowtr( ) - Write RECID Variable on page 699

prowtd( ) - Write DATE Variable

The `prowtd( )` function writes a date value to a shared DATE variable.

On successful completion, `prowtd( )` returns 0; otherwise, it returns a non-zero value:

Syntax

```
int
prowtd ( psymnam, index, year, month, day, unknown )
  char *psymnam;
  int   index;
  int   year;
  int   month;
  int   day;
  int   unknown;
```

psymnam

This input parameter points to the name of the specified shared variable. You supply the name from your OpenEdge application.

index

This input parameter specifies an index value for an array variable. If the variable is not an array, you must set the value of index to 0.

year

This input parameter contains the year value for the specified DATE variable.

month

This input parameter contains the month value for the specified DATE variable.

day

This input parameter contains the day value for the specified DATE variable.

unknown

If this input parameter contains the value 1, the Unknown value (?) is assigned to the variable. If this input parameter contains the value 0, the input value contained in year, month, and day is assigned to the variable.

See Also:

prowtc( ) - Write CHARACTER Variable on page 695, prowti( ) - Write INTEGER Variable on page 697, prowtl( ) - Write LOGICAL Variable on page 698, prowtn( ) - Write DECIMAL Variable on page 699, prowtr( ) - Write RECID Variable on page 699

prowti( ) - Write INTEGER Variable

The prowti( ) function writes an integer value to a shared INTEGER variable.

On successful completion, prowti( ) returns 0; otherwise, it returns a non-zero value:

Syntax

```
int
prowti ( psymnam, index, var, unknown )
  char *psymnam;
  int   index;
  long  var;
  int   unknown;
```

psymnam

This input parameter points to the name of the specified shared variable. You supply the name from your OpenEdge application.

index

This input parameter specifies an index value for an array variable. If the variable is not an array, you must set the value of index to 0.

var

This input parameter contains the integer value for the specified INTEGER variable.

unknown

If this input parameter contains the value 1, the Unknown value (?) is assigned to the variable. If this input parameter contains the value 0, the input value contained in var is assigned to the variable.

See Also:

prowtc( ) - Write CHARACTER Variable on page 695, prowtd( ) - Write DATE Variable on page 696, prowtl( ) - Write LOGICAL Variable on page 698, prowtn( ) - Write DECIMAL Variable on page 699, prowtr( ) - Write RECID Variable on page 699

prowtl( ) - Write LOGICAL Variable

The `prowtl( )` function writes a Boolean value to a shared `LOGICAL` variable.

On successful completion, `prowtl( )` returns 0; otherwise, it returns a non-zero value:

Syntax

```
int
prowtl ( psymnam, index, var, unknown )
  char *psymnam;
  int   index;
  int   var;
  int   unknown;
```

psymnam

This input parameter points to the name of the specified shared variable. You supply the name from your OpenEdge application.

index

This input parameter specifies an index value for an array variable. If the variable is not an array, you must set the value of index to 0.

var

This input parameter contains the boolean value for the specified `LOGICAL` variable. Set var to 1 if you want to assign the value of the variable to `TRUE`; set it to 0 otherwise.

unknown

If this input parameter contains the value 1, the Unknown value (?) is assigned to the variable. If this input parameter contains the value 0, the input value contained in var is assigned to the variable.

See Also:

prowtc( ) - Write CHARACTER Variable on page 695, prowtd( ) - Write DATE Variable on page 696, prowti( ) - Write INTEGER Variable on page 697, prowtn( ) - Write DECIMAL Variable on page 699, prowtr( ) - Write RECID Variable on page 699

prowtn( ) - Write DECIMAL Variable

The `prowtn( )` function writes a decimal value, formatted as a character string, to a shared `DECIMAL` numeric variable.

On successful completion, `prowtn( )` returns 0; otherwise, it returns a non-zero value:

Syntax

```
int
prowtn ( psymnam, index, pvar, unknown )
  char *psymnam;
  int   index;
  char *pvar;
  int   unknown;
```

psymnam

This input parameter points to the name of the specified shared variable. You supply the name from your OpenEdge application.

index

This input parameter specifies an index value for an array variable. If the variable is not an array, you must set the value of index to 0.

pvar

This input parameter points to a buffer containing the character string value for the specified `DECIMAL` numeric variable. The character string must be null terminated.

unknown

If this input parameter contains the value 1, the Unknown value (?) is assigned to the variable. If this input parameter contains the value 0, the input value contained in pvar is assigned to the variable.

See Also:

prowtc( ) - Write CHARACTER Variable on page 695, prowtd( ) - Write DATE Variable on page 696, prowti( ) - Write INTEGER Variable on page 697, prowtl( ) - Write LOGICAL Variable on page 698, prowtr( ) - Write RECID Variable on page 699

prowtr( ) - Write RECID Variable

The `prowtr( )` function writes an integer value to a shared `RECID` variable.

On successful completion, `prowtr( )` returns 0; otherwise, it returns a non-zero value:

Syntax

```
int
prowtr ( psymnam, index, var, unknown )
  char *psymnam;
  int   index;
  long  var;
  int   unknown;
```

psymnam

This input parameter points to the name of the specified shared variable. You supply the name from your OpenEdge application.

index

This input parameter specifies an index value for an array variable. If the variable is not an array, you must set the value of index to 0.

var

This input parameter contains the integer value for the specified RECID variable.

unknown

If this input parameter contains the value 1, the Unknown value (?) is assigned to the variable. If this input parameter contains the value 0, the input value contained in var is assigned to the variable.

See Also:

prowtc( ) - Write CHARACTER Variable on page 695, prowtd( ) - Write DATE Variable on page 696, prowti( ) - Write INTEGER Variable on page 697, prowtl( ) - Write LOGICAL Variable on page 698, prowtn( ) - Write DECIMAL Variable on page 699

E

Command and Utility Reference

This appendix contains the standard reference documentation for QUOTER utility on page 701.

For details, see the following topics:

- QUOTER utility

QUOTER utility

The QUOTER utility formats character data in a file to the standard format so it can be used by an ABL (Advanced Business Language) procedure. By default, QUOTER does the following:

- Takes the name of a file as an argument
- Reads input from that file
- Places quotes (") at the beginning and end of each line in the file
- Replaces any already existing quotes (") in the data with two quotes (" ")

You can use the QUOTER utility directly from the operating system using the operating system statement, as appropriate, to reformat data from a file that is not in the standard ABL input format:

Operating system	Syntax
UNIX Windows	quoter *input-file-name* [> *output-file-name* | -d *character* | -c *startcol-stopcol*]...

input-file-name

Specifies the file you are using.

> *output-file-name*

Specifies the file where you want to send the output.

-d *character*

Identifies a field delimiter. You can specify any punctuation character.

-c *startcol-stopcol*

Specifies the ranges of column numbers to be read as fields. Do not use any spaces in the range list.

Suppose your data file looks like i-datf12.d.

i-datf12.d

```
90
Wind Chill Hockey
BBB
91
Low Key Checkers
DKP
92
Bing's Ping Pong
SLS
```

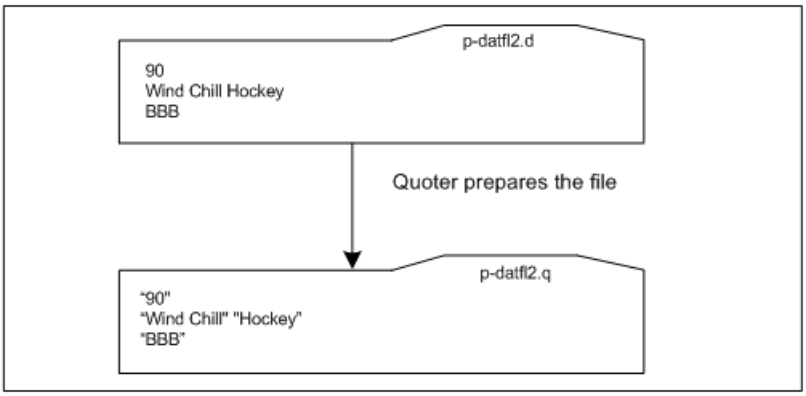
You use QUOTER to put this file into standard format. Use commands shown in the following table to run QUOTER from the Procedure Editor.

Table 86: QUOTER examples

Operating system	Using QUOTER—Examples
Windows	DOS quoter i-dat12.d >i-dat12.q
UNIX	UNIX quoter i-datf12.d >i-datf12.q

In the following figure, `i-datfl2.d` is the name of the data file you supply to `QUOTER` while `i-datfl2.q` is the name of the file in which you want to store `QUOTER`'s output.

Figure 69: How QUOTER prepares a file



The `i-datfl2.q` file contains the `QUOTER` output.

i-datf12.q

```
"90"
"Wind Chill Hockey"
"BBB"
"91"
"Low Key Checkers"
"DKP"
"92"
"Bing's Ping Pong"
"SLS"
```

Now this file is in the appropriate format to be used as input to an ABL procedure.

What if each of the field values in your data file is not on a separate line (unlike `i-datfl2.d`) and without quotes (unlike `i-datf1.d`)? That is, your data file looks like `i-datfl3.d`.

i-datf13.d

```
90 Wind Chill Hockey  BBB
91 Low Key Checkers   DKP
92 Bing's Ping Pong   SLS
```

Suppose you wanted to use this file as the input source to create customer records for customers `90`, `91`, and `92`. You need to make each line of the file into a character string and then assign a substring of this value to each field. The procedure `i-chgin3.p` does that.

i-chgin3.p

```
       DEFINE VARIABLE data AS CHARACTER NO-UNDO FORMAT "x(78)".

/*1*/  OS-COMMAND SILENT quoter i-datfl3.d > i-datfl3.q.
/*2*/  INPUT FROM i-datfl3.q NO-ECHO.

       REPEAT:
/*3*/    CREATE Customer.
/*4*/    SET data WITH NO-BOX NO-LABELS WIDTH-CHARS 80.
         ASSIGN
           Customer.CustNum  = INTEGER(SUBSTRING(data,1,2))
           Customer.Name     = SUBSTRING(data,4,17)
/*5*/      Customer.SalesRep = SUBSTRING(data,22,3).
       END.

/*6*/  INPUT CLOSE.
```

The numbers to the left of the procedure correspond to the following step-by-step descriptions:

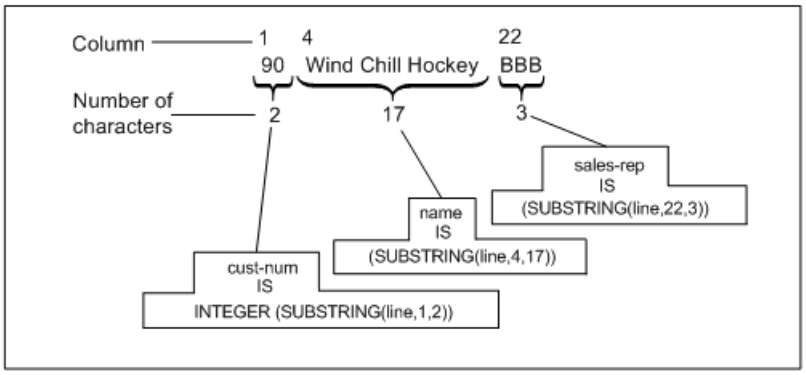
1. The QUOTER utility takes the data from the `i-datfl3.d` file and produces data that looks like the `i-datf13.q` example.

   i-datf13.q

   ```
   "90 Wind Chill Hockey  BBB"
   "91 Low Key Checkers   DKP"
   "92 Bing's Ping Pong   SLS"
   ```

2. The `INPUT FROM` statement redirects the input stream to get input from the `i-datfl3.q` file.
3. The `CREATE` statement creates an empty customer record.
4. The `SET` statement uses the first quoted line in the `i-datfl3.q` file as input and puts that line in the data variable. Once that line of data is in the line variable, the next statements break it up into pieces that get stored in individual customer fields.
5. The `SUBSTRING` functions take the appropriate pieces of the data in the line variable and store the data in the `cust–num`, `name`, and `sales–rep` fields, as shown in the following figure.

   Figure 70: Extracting QUOTER input with SUBSTRING

   Because ABL assumes that all the data in the `i-datfl3.q` file is character data, you must use the `INTEGER` function to convert the `cust–num` data to an integer value.

6. The `INPUT CLOSE` statement closes the input stream coming from the `i-datfl3.q` file and redirects the input stream to the terminal.

Note: With this method, all trailing blanks are stored in the database. To avoid this problem, use the -c or -d option of QUOTER.

You can use QUOTER to prepare files formatted in other ways as well. For example, suppose the field values in your data file are separated by a specific character, such as a comma (,), as in i-datfl4.d.

i-datf14.d

```
90,Wind Chill Hockey,BBB
91,Low Key Checkers,DKP
92,Bing's Ping Pong,SLS
```

You can use a special option, -d, (on UNIX) to tell QUOTER what character separates fields. The procedure i-chgin4.p reads the comma-separated data from i-datfl4.d.

i-chgin4.p

```
OS-COMMAND SILENT quoter -d , i-datfl4.d >i-datfl4.q.

INPUT FROM i-datfl4.q NO-ECHO.
REPEAT:
  CREATE Customer.
  SET Customer.CustNum Customer.Name Customer.SalesRep.
END.
INPUT CLOSE.
```

Here, the -d option or the DELIMITER qualifier tells QUOTER that a comma (,) is the delimiter between each field in the data file. The output of QUOTER is shown in i-datfl4.q.

i-datf14.q

```
"90" "Wind Chill Hockey" "BBB"
"91" "Low Key Checkers"  "DKP"
"92" "Bing's Ping Pong"  "SLS"
```

This data file is in the standard blank-delimited ABL format. If your data file does not use a special field delimiter that you can specify with the -d QUOTER option or the /DELIMITER qualifier, but does have each data item in a fixed column position, you can use another special option, -c, on Windows and UNIX.

You use the -c option or /COLUMNS to identify the columns in which fields begin and end. For example, suppose your file looks like i-datfl5.d.

i-datf15.d

```
90 Wind Chill Hockey BBB
91 Low Key Checkers  DKP
92 Bing's Ping Pong  SLS
```

The procedure i-chgin5.p uses this data file to create customer records.

i-chgin5.p

```
OS-COMMAND SILENT quoter -c 1-2,4-20,22-24 i-datfl5.d >i-datfl5.q.

INPUT FROM i-datfl5.q NO-ECHO.
REPEAT:
  CREATE Customer.
  SET Customer.CustNum Customer.Name Customer.SalesRep.
END.
INPUT CLOSE.
```

Because you used the `-c` option, this procedure produces a data file without trailing blanks.

You can also use `QUOTER` interactively to reformat your data. You can access `QUOTER` interactively through the Administration Tool or, in character interfaces, the Data Dictionary. From the main menu, choose **Utilities**.

F

Database Administration Entity Interface Reference

This appendix provides a reference to the entity interfaces that support the ABL API for Multi-tenant and Table Partitioning Maintenance. For more information on this API, see ABL API for Multi-tenant and Table Partition Management on page 205. The entity interfaces provide public access to classes that map to entities in the database through properties that correspond to key fields. Some properties have more user friendly names than the database field they are mapped to, while others have names that are similar to the database field names, but without underscores and hyphens. In some cases, properties are implemented with a different data type than the underlying data source. Fields that represent foreign keys for relationships are implemented as objects.

Each interface instance has a key that uniquely identifies the entity in the API and maps it to the corresponding row in the database. The API attempts to expose the humanly readable character key whenever possible, but internal IDs are also typically exposed as properties. Some entity interfaces, like `IPartition`, do not expose the internal key.

All entity class implementations override the `Equals( )` method to account for the fact that different object instances can represent the same database data and return `TRUE` if the passed object represents the same database record and is retrieved by the same service.

All entity interfaces (and classes) inherit (or implement) the `IDataAdminElement` interface, and in some cases, other interfaces. In the sections that describe these interfaces, the interface names are referenced without their containing package, which is `OpenEdge.DataAdmin` for entity interfaces and `OpenEdge.DataAdmin.Lang.Collections` for entity collection interfaces (documented in Database Administration Entity Collection Interface Reference on page 773).

The following sections have code examples that show how to use the interfaces and classes with the service. Note that these code examples largely assume the `USING` statement declarations for `OpenEdge.DataAdmin.*` and `OpenEdge.DataAdmin.Lang.Collections.*`, and the definition and instantiation of the DataAdminService (`service`) as shown in Instantiating a DataAdminService on page 213.

For details, see the following topics:

- IAdministrator interface
- IArea interface
- IAuthenticationSystem interface
- IDataAdminElement interface
- IDomain interface
- IExtent interface
- IField interface
- IFieldPermission interface
- IIndex interface
- IIndexField interface
- IPartition interface
- IPartitionOwner interface
- IPartitionParent interface
- IPartitionPolicy interface
- IPartitionPolicyDetail interface
- ISchema interface
- ISchemaElement interface
- ISequence interface
- ISequenceValue interface
- ITable interface
- ITablePermission interface
- ITenant interface
- ITenantGroup interface
- IUser interface
- IUserFieldPermission interface
- IUserPermission interface
- IUserTablePermission interface

IAdministrator interface

The IAdministrator interface provides a single Administrators property which identifies the users defined as database Security Administrators for the database.

IAdministrator properties

Inherits properties from IDataAdminElement and defines the properties listed in the following table.

Table 87: Multi-tenant Maintenance IAdministrator interface properties

Property name	Data type	Access	Description
Administrators	CHARACTER	Readable, Writeable	Specifies the users defined as Database Administrators using a CAN-DO user ID permissions expression. For more information, see the CAN-DO function entry in *ABL Reference* and the sections on user ID patterns and ABL permissions checking in *Introduction to Identity Management*.

IAdministrator methods

Inherits methods from IDataAdminElement.

IAdministrator examples

The following code updates the Security Administrators for the database:

```
define variable admin as IAdministrator no-undo.

admin = service:GetAdministrator( ).
admin:Administrators = "xxx@*".
service:UpdateAdministrator(admin).
```

The GetAdministrator( ) method in the service has no parameter and returns a single object that applies to the entire database. The pattern, "xxx@*", is a user id pattern, which can be formed following the rules of an ID pattern list as defined for the ABL built-in CAN-DO function. For more information, see the CAN-DO function entry in *ABL Reference*, and see the sections on user ID patterns and ABL permissions checking in *Introduction to Identity Management*.

See also:

- IDataAdminElement interface on page 716

IArea interface

The IArea interface allows you to work with areas without editing a structure description (.st) file.

The underlying implementation generates a .st file and passes it as a parameter to the PROSTRCT utility in order to add it to the database the service is managing.

Note: The current implementation supports a minimum of validation. Most error conditions are caught when the underlying PROSTRCT utility is executed. This error message might include information about where in the .st file the error occurred and also indicate that "prostrct failed". Some of this error information might appear irrelevant or confusing because the .st file on which PROSTRCT operates is generated by the API. Future versions will likely have more validation at an earlier stage and not include the specific PROSTRCT details in messages.

IArea properties

Inherits properties from IDataAdminElement and defines the properties listed in the following table.

Table 88: Multi-tenant Maintenance IArea interface properties

Property name	Data type	Access	Description
BlockSize	INTEGER	Read-only	Returns the block size of the area in KB.
ClusterSize	INTEGER	Readable, Writeable	Set to the cluster size. Can be 1 (for a Type 1 area), 8, 64, or 512.
Extents	IExtentList	Read-only	Returns a numbered list of the area's extents.
IsType2	LOGICAL	Read-only	Returns TRUE if the area is a Type 2 area.
Name	CHARACTER	Readable, Writeable	Set to the unique name and identifier (in the database) of the area.
Number	INTEGER	Read-only	Returns the internal unique area number.
NumExtents	INTEGER	Read-only	Returns the number of extents.

Property name	Data type	Access	Description
RecordsPerBlock	INTEGER	Readable, Writeable	Records per block. Can be set to 1, 2, 4, 8, 16, 32, 64, 128, or 256.
Type	CHARACTER	Read-only	Returns the area type, which can only be "Data" for a new area created using this API. You can create other possible area types using the database PROSTRCT utility. For more information, see *Manage the OpenEdge Database*.

IArea methods

Inherits methods from IDataAdminElement.

IArea examples

The following sections provide examples of using IArea.

Creating areas

The following code shows assignment of properties and extents for a new area. There need to be at least one extent in the area before it can be passed to the service to be created.

The ClusterSize and RecordsPerBlock properties both defaults to 64 and does not need be assigned even if they are mandatory:

```
define variable area   as IArea no-undo.
define variable extent as IExtent no-undo.

assign
  area = service:NewArea("OrderDataArea")
  area:ClusterSize = 8
  area:RecordsPerBlock = 32
  extent = service:NewExtent()
  extent:Size = 1024
  extent:Path = "C:\db".

area:Extents:Add(extent).

service:CreateArea(area).
```

Updating areas used for extents

There are no properties that can be edited on an existing area, but extents can be added later and committed to the service as follows:

```
area = service:GetArea("OrderDataArea")
extent = new Extent()
extent:Path = "C:\db".
area:Extents:Add(extent).
service:UpdateArea(area).
```

See also:

- IAreaSet interface on page 775
- IDataAdminElement interface on page 716
- IExtentList interface on page 781

IAuthenticationSystem interface

The `IAuthenticationSystem` interface provides the properties and methods needed to create and edit records in the `_Sec-authentication-system` database table. This table defines authentication systems that can be used to authenticate users for access to the database. For more information on authentication systems in OpenEdge, see *Introduction to Identity Management*.

IAuthenticationSystem properties

Inherits properties from `IDataAdminElement` and defines the properties listed in the following table.

Table 89: Multi-tenant Maintenance IAuthenticationSystem interface properties

Property name	Data type	Access	Description
`Callback`	`CHARACTER`	Readable, Writeable	Specifies the filename of an ABL external procedure (with or without the extension) that defines one or both of the following callbacks: • `AuthenticateUser` • `AfterSetUser` These callbacks support user authentication and single sign-on (respectively) in different ways depending on the type of authentication system. For more information, see the description of the `IsEnabled` property of this interface and the sections on using ABL callbacks in authentication systems in Application Security on page 97

Property name	Data type	Access	Description
IsEnabled	LOGICAL	Readable, Writeable	Set to TRUE if the authentication system is enabled for user authentication, and set to FALSE if the authentication system is enabled for single sign-on (SSO) only. For a user-defined authentication system, a TRUE setting indicates that any ABL external procedure specified by the Callback property implements the code to authenticate users against a secure user account system for an OpenEdge-performed user authentication operation, and optionally extends an SSO operation. For a supported OpenEdge built-in authentication system, a TRUE setting indicates that any ABL external procedure specified by the Callback property extends the OpenEdge-performed user authentication operation already implemented by the authentication system, and optionally extends an SSO operation. A FALSE setting indicates that the procedure only extends an SSO operation for a supported authentication system. For more information, see the sections on using ABL callbacks in authentication systems in Application Security on page 97. **Note:** This property is writeable only for a user-defined authentication system. You cannot change this value for an OpenEdge built-in authentication system.
Name	CHARACTER	Readable, Writeable	Set to a unique name and identifier (within the database) of the authentication system.

Property name	Data type	Access	Description
Description	CHARACTER	Readable, Writeable	Set to a free text description that describes the authentication system, such as its purpose, use, etc., primarily for tools and reporting purposes.
Comments	CHARACTER	Readable, Writeable	Set to custom specific information about this authentication system.

IAuthenticationSystem methods

Inherits methods from IDataAdminElement.

IAuthenticationSystem examples

Creating an authentication system

The following code creates a new authentication system:

```
define variable authSys as IAuthenticationSystem no-undo.

assign
  authSys = service:NewAuthenticationSystem("MyAuthenticationSystem")
  authSys:Description = "This auth system is for xx users."
  authSys:Callback = "myauthproc.p"
  authSys:IsEnabled = true.

service: CreateAuthenticationSystem(authSys).
```

Updating an existing authentication system

The following code edits a property on an existing authentication system:

```
assign
  authSys = service:GetAuthenticationSystem("MyAuthenticationSystem")
  authSys:Description = "This authentication system is ok.".

service:UpdateAuthenticationSystem(authSys).
```

See also:

- IAuthenticationSystemSet interface on page 776
- IDataAdminElement interface on page 716

IDataAdminElement interface

The IDataAdminElement interface defines properties and methods inherited by all other entity interfaces described in this section. These properties and methods are provided for application use on a sub-interface. Otherwise, the IDataAdminElement interface is not supported for direct access.

Note also that this interface defines several transient properties that reflect the current state of the derived object. The interface defines no properties that persist in the database.

The properties and methods described in the following sections are supported for use in your application code. IDataAdminElement also defines other properties and methods not described here that are reserved for OpenEdge use only.

IDataAdminElement properties

Defines the properties listed in the following table.

Table 90: Multi-tenant Maintenance IDataAdminElement interface properties

Property name	Data type	Access	Description
Attached	LOGICAL	Read-only	Returns TRUE if the instance is in a service or collection.
ContextId	CHARACTER	Read-only	Returns the context for the instance. Used in the Equals( ) method to ensure that compared entities with the same keys and underlying data are identified as equal. A blank value means the object has no context (yet), is new, and is not attached, in which case Equals( ) is only TRUE if it compares object references to the same IDataAdminElement instance.
RequestInfo	IRequestInfo	Read-only	Returns an optional definition for unique query criteria to find this object and any specified child collections. If no IRequestInfo object is assigned, this property has the Unknown value (?). For more information on using the IRequestInfo interface, see Filtering on a child collection using the RequestInfo class on page 228.
Service	IDataAdminService	Read-only	Returns the DataAdminService from which the object was retrieved.

IDataAdminElement methods

Defines the methods listed in the following table.

Table 91: Multi-tenant Maintenance IDataAdminElement interface methods

Method syntax	Description
`VOID Export( )`	Exports the entity data to a JSON file named for the sub-interface entity, for example, `area.json` or `tenant.json` for an `IArea` or `ITenant` object, respectively.[13]
`VOID Export ( cFilename AS CHARACTER )`	Export entity data to the file specified by *`cFilename`*[13]
`VOID ExportTree ( cFilename AS CHARACTER )`	Export entity data with all child collections to the file specified by *`cFilename`*.[14]
`VOID ExportTree ( cFilename AS CHARACTER , pcCollections AS CHARACTER )`	Export entity data with specified child collections to the file specified by *`cFilename`*.[14]
`VOID Import ( cFilename AS CHARACTER )`	Import entity data from the file specified by *`cFilename`*. (The file must exist.)[13]
`VOID ImportTree ( cFilename AS CHARACTER )`	Import entity data, including child collections, from file specified by *`cFilename`*. (The top-level entity must exist.)[13]

IDataAdminElement examples

For more information on using the `Export( )` and `Import( )` methods, see Exporting and importing on page 230. For examples of using the `RequestInfo` property, see Filtering on a child collection using the RequestInfo class on page 228.

See also:

- IDataAdminCollection interface on page 777

[13] Currently supports JSON (.json) files only.

[14] Currently supports JSON (.json) files only, except for the ITenant and ITenantGroup sub-interfaces, which also support the generation of ABL procedure (.p) files.

IDomain interface

The IDomain interface provides the properties and methods to create and edit records in the _Sec-authentication-domain database table. In addition to regular data properties IDomain provides a Users property that returns a collection of users for the domain. This can be used to add or delete users in the domain.

IDomain properties

Inherits properties from IDataAdminElement and defines the properties listed in the following table.

Table 92: Multi-tenant Maintenance IDomain interface properties

Property name	Data type	Access	Description
AccessCode	CHARACTER	Readable, Writeable	Set to an access code that OpenEdge automatically encrypts and uses to seal any client-principal whose user credentials are authenticated to this domain. The encrypted value is also used to validate the identity and integrity of any client-principal object that is sealed when a user authenticates to the domain. Any value you set can be a case-sensitive string consisting of printable characters (including embedded spaces and tabs) with a null terminator. The OpenEdge encryption generates a MAC value for sealing and validating client-principal objects.
AuditingContext	CHARACTER	Readable, Writeable	Set to the audit context value when a _Client-session record is recorded in the database. It is free text that typically describes the domain's authentication system, location, and any other identifying characteristics. While this value is optional for auditing, it is recommended that a value be filled in.
AuthenticationSystem	IAuthenticationSystem	Readable, Writeable	Set to an entity object that defines the authentication system configured for the domain.
Comments	CHARACTER	Readable, Writeable	Set to free-text comments about the domain, such as customized authentication details.

Property name	Data type	Access	Description
Description	CHARACTER	Readable, Writeable	Set to a free-text description of the domain primarily for reporting purposes.
IsBuiltin	LOGICAL	Read-only	Returns TRUE if the domain is an OpenEdge built-in domain pre-configured in OpenEdge database.
IsEnabled	LOGICAL	Readable, Writeable	Set to TRUE to enable the domain, and set to FALSE to disable the domain without removing it from the database.
Name	CHARACTER	Readable, Writeable	Set to the unique name and identifier (within a database) of a security domain that can be used to authenticate or validate a user's identity. A security domain identifies the authentication system used to authenticate users in that domain. ABL or SQL refers to this name when it performs a user authentication operation or validates the client-principal object (security token) that is sealed by an authentication operation performed in this domain. In a multi-tenant database environment, the domain name must be used to fully qualify a user's identity by appending the domain name to the user name using an '@' delimiter, for example, "fred@ldap". The user must provide this fully qualified user ID for all user authentication operations in a multi-tenant environment.
RuntimeOptions	CHARACTER	Readable, Writeable	Set to a comma-delimited list of run-time options that depend on the authentication system.
SystemOptions	CHARACTER	Readable, Writeable	Set to any PAM options that you need to provide to the authentication system during user authentication.

Property name	Data type	Access	Description
Tenant	ITenant	Readable, Writeable	Set to an entity object that identifies the tenant a user can access through the domain.
Users	IUserSet	Read-only	Returns a collection of the OpenEdge users defined in the _User table with this domain. If the domain is not configured for the _oeusertable authentication system, this property returns an empty collection.

IDomain methods

Inherits methods from IDataAdminElement.

IDomain examples

Creating a new domain

The following code shows assignment of properties for a new domain. The Tenant is a mandatory property and has to be defined before the domain can be created:

```
define variable domain as IDomain no-undo.

assign
     domain = service:NewDomain("AcmeDomain")
  domain:AuthenticationSystem  = service:GetAuthenticationSystem("_oeusertable")
  domain:AccessCode  = <user variable>
  domain:Tenant      = service:GetTenant("Acme")
  domain:Description = "This domain is the default domain for Acme.").
service:CreateDomain(domain).
```

Editing properties on an existing domain

The following code edits one of the properties on an existing domain:

```
assign
  domain = service:GetDomain("AcmeDomain")
  domain:Description = "This domain is the only domain for Acme.".
service:UpdateDomain(domain).
```

Adding users to an existing domain

The following code shows how to use the `Users` property to add new users to an existing domain. This can also be done when the domain is created. This code shows the usage of the `NewUser( )` factory method, passing only the name part of the user ID. The `IUser` interface `Domain` property and `Tenant` property are updated automatically when adding the user to the `IDomain` user collection through its `Users` property. For more information, see IUser interface on page 766:

```
define variable domain as IDomain no-undo.
define variable user   as IUser   no-undo.

domain = service:GetDomain("AcmeDomain").

assign
  user = service:NewUser("randersen")
  user:GivenName = "Robert"
  user:SurName = "Andersen".

domain:Users:Add(user).

assign
  user = service:NewUser("mandersen")
  user:GivenName = "Molly"
  user:SurName = "Andersen".
domain:Users:Add(user).

service:UpdateDomain(domain).
```

See also:

- IAuthenticationSystem interface on page 712
- IDataAdminElement interface on page 716
- IDomainSet interface on page 780
- ITenant interface on page 755
- IUserSet interface on page 801

IExtent interface

The `IExtent` interfaces enables you to work with areas without editing a structure description (`.st`) file.

The underlying implementation generates a `.st` file and passes it as parameter to the PROSTRCT utility in order to add it to the database that the service is managing.

IExtent properties

Inherits properties from `IDataAdminElement` and defines the properties listed in the following table.

Table 93: Multi-tenant maintenance IExtent interface properties

Property name	Data type	Access	Description
`Area`	`IArea`	Read-only	Returns the area of the extent.
`FileName`	`CHARACTER`	Read-only	Returns the filename of the extent.
`IsFixed`	`LOGICAL`	Readable, Writeable	Set to `TRUE` if the extent is fixed.
`Number`	`INTEGER`	Read-only	Returns the number of the extent within the area.
`Path`	`CHARACTER`	Readable, Writeable	Set to the path of the extent. Can be set instead of the `FileName` property.
`Size`	`INTEGER`	Readable, Writeable	Set to the size of the extent in KB.
`SizeUsed`	`INTEGER`	Read-only	Returns the used extent size (high water mark) in KB.

IExtent methods

Inherits methods from `IDataAdminElement`.

IExtent examples

For more information and examples, see the code examples in IArea interface on page 710.

See also:

- IArea interface on page 710
- IDataAdminElement interface on page 716
- IExtentList interface on page 781

IField interface

The `IField` interface is mapped to the `_Field` table in the schema, but only exposes a few properties needed for multi-tenancy and table partitioning purposes.

It inherits the `ISchemaElement` interface, which is also a super interface for the `ITable` and `IIndex` interfaces and allows the `IPartition` interface to have a single property that represents the `ITable`, `IIndex`, or `IField` to which it belongs.

You can use the `Partitions` property inherited from `ISchemaElement` to access field partitions for all tenants and tenant groups.

Note:

The API only returns LOB fields in collections and expects LOB fields as keys in partition maps.

The only writeable property on field partitions is the `Area` property.

IField properties

Inherits properties from `ISchemaElement` and defines the properties listed in the following table.

Table 94: Multi-tenant Maintenance IField interface properties

Property name	Data type	Access	Description
`DataType`	`CHARACTER`	Read-only	Returns the data type of the field.
`Table`	`ITable`	Read-only	Returns the parent table of the field.
`IsLob`	`LOGICAL`	Read-only	Returns TRUE if the field is a LOB field. This is true when the data type is either "CLOB" or "BLOB".

IField methods

Inherits methods from `ISchemaElement`.

See also:

- IFieldSet interface on page 782
- IIndex interface on page 725
- IPartition interface on page 727
- ISchemaElement interface on page 746
- ITable interface on page 751

IFieldPermission interface

The `IFieldPermission` interface is mapped to the `_Field` table in the schema, but it only exposes the `_Can*` fields that define who has read and write permissions to the field.

The `DataAdminService` does not provide any CRUD methods for field permissions as they are only exposed as a collection property on `ITablePermission`.

IFieldPermission properties

Inherits properties from `IDataAdminElement` and defines the properties listed in the following table.

Table 95: Multi-tenant Maintenance IFieldPermission interface properties

Property name	Data type	Access	Description
`CanRead`	`CHARACTER`	Readable, Writeable	Set to a `CAN-DO` user ID permissions expression that defines who can read the field.[15]
`CanWrite`	`CHARACTER`	Readable, Writeable	Set to a `CAN-DO` user ID permissions expression that defines who can write to the field.[15]
`Name`	`CHARACTER`	Readable, Writeable	Set to the name of the field.
`Table`	`ITable`	Read-only	Returns the table to which the field permissions apply.

IFieldPermission methods

Inherits methods from `IDataAdminElement`.

IFieldPermission examples

The following code retrieves an `ITablePermission` object from the service, edits one of the `IFieldPermission` objects, and passes the `ITablePermission` back to the service to be committed to the database:

```
define variable tblPerm as ITablePermission no-undo.
define variable fldPerm as IFieldPermission no-undo.
define variable domain as IDomain no-undo.

tblPerm = service:GetTablePermission("Customer").
fldPerm = tblPerm:FieldPermissions:Find("CustNum").
fldPerm:CanWrite =  "xx@ABCDomain".
service:UpdateTablePermission(tblPerm).
```

See also:

- IDataAdminElement interface on page 716
- IFieldPermissionSet interface on page 782
- ITable interface on page 751
- ITablePermission interface on page 754

[15] For more information, see the CAN-DO function entry in *ABL Reference* and the sections on user ID patterns and ABL permissions checking in *Introduction to Identity Management*.

IIndex interface

The IIndex interface is mapped to the _Index table in the schema, and exposes both read-only and writeable properties needed for multi-tenancy and table partitioning purposes.

It inherits the ISchemaElement interface, which also is a super interface for the ITable and IField interfaces and allows the IPartition interface to have a single property that represents the ITable, IIndex, or IField to which it belongs.

You can use the Partitions property inherited from ISchemaElement to access index partitions for all tenants and tenant groups.

IIndex properties

Inherits properties from ISchemaElement and defines the properties listed in the following table.

Table 96: Multi-tenant Maintenance and Table Partition IIndex interface properties

Property name	Data type	Access	Description
Description	CHARACTER	Read-only	Returns a description of the index.
Fields	IFIELDSET	Read-only	Returns a collection containing the fields in the index. The set may not be in the order the fields are defined in the database. The IndexFields property must be used to get the correct order of the fields.
IActive	LOGICAL	Writeable	Returns TRUE if the index is active.
IsPrimary	LOGICAL	Writeable	Returns TRUE if the index is the primary index for the table.
IsUnique	LOGICAL	Writeable	Returns TRUE if the index is unique.
IsWordIndex	LOGICAL	Writeable	Returns TRUE if the index is a word index.
Table	ITable	Read-only	Returns the parent table of the index.
IsLocal	LOGICAL	Writeable	Returns TRUE if the index is local per partition. (Default = FALSE) This property cannot be edited after the index is created.
IndexFields	IINDEXFIELDLIST	Read-only	Returns a list of the table's index fields. This list ensures that IIndexField elements are listed in the correct order.

IIndex methods

Inherits methods from `ISchemaElement`.

See also:

- IField interface on page 722
- IIndexSet interface on page 783
- IPartition interface on page 727
- ISchemaElement interface on page 746
- ITable interface on page 751

IIndexField interface

The `IIndexField` interface is mapped to the `_Index-field` table in the schema, and exposes only read-only properties needed for table partitioning purposes.

It inherits the `IDataAdminElement` interface.

IIndexField properties

Inherits properties from `IDataAdminElement` and defines the properties listed in the following table.

Table 97: Table Partition IIndexField interface properties

Property name	Data type	Access	Description
`Fields`	`IFIELD`	Read-only	Returns the fields in the index.
`Index`	`IINDEX`	Read-only	Returns the (parent) index of the index field.
`Number`	`INTEGER`	Read-only	Returns the ordinal number of the field in the index.
`IsAscending`	`LOGICAL`	Read-only	Returns `TRUE` if the value of the field must be sorted in an ascending order.
`IsAbbreviate`	`LOGICAL`	Read-only	Returns `TRUE` if this value of the field value is defined as abbreviated.

See also:

- IField interface on page 722
- IIndexSet interface on page 783
- IPartition interface on page 727

- ISchemaElement interface on page 746
- ITable interface on page 751

IPartition interface

The IPartition interface is a logical representation of a physical storage partition and exists even if the partition is not physically allocated. It is mapped to the _Storage-object record in the database and is thus never explicitly created or deleted by the service. Thus, partition collections do not support an Add( ) or Remove( ) method. There is always one partition for each

- Tenant group
- Tenant that is not part of a group
- Multi-tenant enabled table index and LOB field in the database

IPartition is an exception among most other IDataAdminElement interfaces in that it does not have a single scalar key. IPartition has a new Owner property of type IPartitionOwner, allowing the interface to define the relation to ITenant, ITenantGroup, ITable, IField, IIndex, and IpartitionPolicyDetail with the same key property. The collection interfaces through which partitions can be accessed using these properties include, IPartitionCollection, IPartitionMap, and ISchemaPartitionMap.

The DataAdminService does not provide any CRUD methods for partitions alone, as partitions always exist to support other entities.

IPartition has an AllocationState property that can be set to "Allocated", "Delayed", or "None" and is initialized based on the value of the DefaultAllocationState property on the ITenant or ITenantGroup or IPartitionPolicyDetail to which it belongs. The "Delayed" setting allows the allocation to be delayed or allocated later using the Allocate( ) method on an ITenant or ITenantGroup or IPartitionPolicyDetail. The "None" setting also postpones the allocation and allows the partition to be allocated individually later (or never). The property setting cannot be changed once it is set to "Allocated".

IPartition has an Allocate( ) method that is the equivalent of setting AllocationState to "Allocated" but in addition returns TRUE or FALSE to indicate whether the allocation has actually completed.

The AllocationState property can only be edited in table partitions. Field and index partitions obtain the setting from table partitions.

Note:

All entity objects that have partitions have Allocate( ) methods that allow allocation of all contained partitions. The default is to allocate partitions set to "Delayed" similar to the database. However, in some cases the API also allows allocation of multiple partitions that are set to "None".

If you want to use an area other than the default, you must also set the Area property on IPartition before the partition is allocated.

IPartition properties

Inherits properties from IDataAdminElement and defines the properties listed in the following table.

Table 98: Multi-tenant and Table Partition Maintenance IPartition interface properties

Property name	Data type	Access	Description
`AllocationState`	`CHARACTER`	Readable, Writeable	Set to the partition's allocation state, a character value equal to `"Allocated"`, `"Delayed"`, or `"None"`. This property is writeable only for table (not field or index) partitions and becomes read-only once the partition is allocated. For more information on these settings, see the description of the `DefaultAllocation` property in ITenant interface on page 755.
`Area`	`IArea`	Readable, Writeable	Set to the area of the partition. This property becomes read-only once the partition is allocated.
`BufferPool`	`CHARACTER`	Readable, Writeable	Set to the partition's buffer pool, a character value equal to `"Primary"` or `"Alternate"`.
`CanAssignAlternateBufferPool`	`LOGICAL`	Read-only	Returns `TRUE` if the `BufferPool` property can be set to `"Alternate"`.
`Element`	`ISchemaElement`	Read-only	Returns the `ITable`, `IField`, or `IIndex` object of the partition as an `ISchemaElement`.
`ObjectType`	`CHARACTER`	Read-only	Returns the object type of the partition, a character value equal to `"Table"`, `"Index"`, or `"Field"`.
`Owner`	`IPartitionOwner`	Readable, Writeable	Sets the relation to ITenant, ITenantGroup or IPartitionPolicyDetail with the same key property.
`Tenant`	`ITenant`	Read-only	Returns the tenant of the partition. The `Unknown value (?)` if this is a tenant group partition.
`TenantGroup`	`ITenantGroup`	Read-only	Returns the tenant group of the partition. The `Unknown value (?)` if this is a tenant partition.

IPartition methods

Inherits methods from `IDataAdminElement` and defines the methods listed in the following table.

Table 99: Multi-tenant Maintenance IPartition interface methods

Method syntax	Description
`LOGICAL Allocate ( )`	Allocates the partition. Throws an error if used on a field or index partition.

IPartition examples

Partition management is described and demonstrated in the following sections:

- Managing partitions for a table on page 753
- Managing partitions for a tenant on page 760
- Managing partitions for a tenant group on page 765

See also:

- IArea interface on page 710
- IDataAdminElement interface on page 716
- IField interface on page 722
- IIndex interface on page 725
- IPartitionCollection interface on page 785
- IPartitionMap interface on page 787
- IPartitionPolicyDetailSet interface on page 789
- ISchemaElement interface on page 746
- ITable interface on page 751
- ITenant interface on page 755
- ITenantGroup interface on page 761

IPartitionOwner interface

The `IPartitionOwner` interface is an interface that inherits the `IPartitionParent interface`.

A new partition is created every time a new instance of this interface is created.

The new `IPartitionPolicy, ITenant, and ITenantGroup` interfaces inherit this.interface.

IPartitionOwner properties

The `IPartitionOwner` interface defines the properties listed in the following table.

Table 100: Table Partition Maintenance IPartitionOwner interface properties

Property name	Data type	Access	Description
`Id`	`INTEGER`	Read-only	Returns the unique, internal tenant identifier that is generated by the database. **Note:** Dump and load generates a new value.
`IsAllocated`	`LOGICAL`	Read-only	Returns TRUE if all table partitions have a state of `"Allocated"` or `"None"`. Returns FALSE if any table partition has a state of `"Delayed"`. **Note:** Use the `Allocate( )` method on `IPartitionParent` to allocate delayed partitions for a table.
`Partitions`	`IPartitionMap`	Read-only	Returns a collection of all partitions of the table partitions mapped by `ISchemaElement` (`ITable`, `IIndex`, and `IField`).

IPartitionOwner methods

Inherits methods from `IPartitionParent` and defines the methods listed in the following table.

Table 101: Table PartitionMaintenance IPartitonParent interface methods

Method syntax	Description
`LOGICAL Allocate ( )`	Allocates all the table's partitions. **Note:** The partition (`IPartitionOwner`) must be passed to a partition `Update*( )` method on the service in order to commit the changes.

See also:

- IDataAdminElement interface on page 716
- IPartitionMap interface on page 787
- IPartitionPolicy interface on page 733
- IPartitionPolicyDetail interface on page 739

- ITenantGroupSet interface on page 797
- ITenantGroup interface on page 761
- IPartitionMap interface on page 787

IPartitionParent interface

The `IPartitionParent` interface abstracts common properties of all interfaces and classes that define partition defaults, and enables and allocates partitions.

The new `IPartitionPolicy` and `IPartitionOwner` interfaces inherit this.interface.

IPartitionParent properties

Inherits properties from `IDataAdminElement` and defines the properties listed in the following table.

Table 102: Table Partition Maintenance IPartitionParent interface properties

Property name	Data type	Access	Description
DefaultAllocation	CHARACTER	Readable, Writeable	Set to a preference that decides whether to allocate partitions when the tenant is created, when tables are multi-tenant enabled, or new multi-tenant tables, indexes, or LOB fields are created. This preference can have the following values: • **"Immediate"** — Allocate immediately using default areas. Allocates all partitions according to the defaults when the tenant is created. • **"Delayed"** — Delay the physical allocation of the partitions. Sets the partitions' AllocateState properties to "Delayed", which allows them to be allocated by executing the Allocate( ) method on the ITenant object. • **"None"** — No automatic allocation. Sets the partitions' AllocateState properties to "None", which means that the allocation can only be done on each individual partition by executing the Allocate( ) method on the IPartition object. **Note:** If you intend to make a tenant a member of a tenant group, set this property to "Delayed" so no storage is allocated to the tenant before you have allocated storage to the group.
DefaultDataArea	IArea	Readable, Writeable	Set to the default data area for new table partitions.
DefaultIndexArea	IArea	Readable, Writeable	Set to the default index area for new table partitions.
DefaultLobArea	IArea	Readable, Writeable	Set to the default LOB area for new table partitions.

Property name	Data type	Access	Description
Description	CHARACTER	Readable, Writeable	Set to a table partition description of your choice.
Name	CHARACTER	Readable, Writeable	Set to a unique table partition name, with a maximum length 32 characters.

See also:

- IDataAdminElement interface on page 716F
- IPartitionPolicy interface on page 733
- IPartitionPolicy interface on page 733
- IPartitionPolicyDetail interface on page 739
- IPartitionMap interface on page 787

IPartitionPolicy interface

The IPartitionPolicy interface is an interface that inherits the IPartitionParent interface and is mapped to the _partition-policy table in the database.

The interface defines properties and methods that are used to create and work with table partitions.

The IPartitionPolicy interface has a unique Name property and the service has dedicated CRUD methods that use the name to identify the policy. The IPartitionPolicy is exposed on the PartitionPolicy property on the ITable, so one can work with policies without referring to them by name.

IPartitionPolicy properties

Inherits properties from IPartitionParent and defines the properties listed in the following tables:

Table 103: IPartitionParent interface inherited properties with different behavior than super class

Property name	Data type	Access	Description
DefaultAllocation	LOGICAL	Readable, Writeable	Valid values are "None" and "Immediate".

Table 104: Table partition policy interface properties

Property name	Data type	Access	Description
HasComposite	LOGICAL	Read-only	Returns TRUE if any partition policy detail has partitions in a composite partition.

Property name	Data type	Access	Description
HasRange	LOGICAL	Readable, Writeable	Set to TRUE (the default) if the policy is a range policy (partitions based on ranges of values). Set to FALSE in order to prevent mapping data to partitions based on a range of a values. You cannot edit the property after the policy has been created in the database.
IsCompositeReadOnly	LOGICAL	Readable, Writeable	Set to TRUE to mark a composite partition as read-only.Note: An error is thrown if you set this property to TRUE for a partition policy without composite partitions.
IsReadOnly	LOGICAL	Readable, Writeable	Set to TRUE to mark a partition policy as read-only.Note: Currently, this property returns FALSE. Setting this property throws an error.
NumFields	INTEGER	Read-only	Returns the number of fields and values that are used to define the partitions of a table.
Table	ITable	Read-only	Returns the parent table of the field. This is editable only when the partition policy is new and has not been assigned to a table's partition policy. You cannot edit the property after the policy has been created in the database.
Fields	IFieldSet	Read-only	Returns a collection containing the fields in the index.The values of the fields are used to define the partitions of the table. The user must add at least one IField to the collection before creating the policy. The collection cannot be edited after the policy is created.
Details	IPartitionPolicyDetailSet	Read-only	Returns a collection containing the details of the partition policy.The values of the policy are used to define the partitions of the table. Policy details along with their values can be added to the collection at any time, as long as the new value does not need to be split.

IPartitionPolicy methods

Inherits methods from `IPartitionPolicyParent`.

Note: The `Table` and `HasRange` properties must be set and the `Fields` collection must have at least one field before calling this method.

Table 105: Table Partition Maintenance IPartitionPolicy interface methods

Method syntax	Description
`AddDetailsFromData()`	Populates the policy details collection from table data.
`AddDetailsFromData(pcTokenString as CHARACTER)`	Populates the policy details collection from table data. The name template for the policy details is specified by `pcTokenString` in the `"${<property(1)>}...-${<property(n)>}"` format where property can be `PartitionPolicyName`, `PartitionPolicyType`, `Number`, `TableName` and `Values[i]` (`i` refers to the ordinal number of the partition fields.) For example, `"${PartitionPolicyName}-${Number}-${Values[1]}"`
`AddDetailsFromData(pcTokenBase as CHARACTER,pcTokenSubstituteArgs as CHARACTER)`	Populates the policy details collection from table data. To specify the name template of the policy details, the `pcTokenBase` parameter sets the template property order in the `"&1...-&n"` format (n can take values up to 9). The `pcTokenSubstituteArgs` parameter specifies comma-separated property values in the order set by `pcTokenBase`. The property values can be `PartitionPolicyName`, `PartitionPolicyType`, `Number`, `TableName` and `Values[i]` (`i` refers to the ordinal number of the partition fields.) For example, `"&1-&2-&3","PartitionPolicyName,Number,Values[1]"`

IPartitionPolicy examples

Creating a partition policy for an empty table

A policy for an empty table can be created without any details. The following code shows the creation of a new partition policy for an empty table:

```
define variable policy as IPartitionPolicy no-undo.
define variable tbl as ITable no-undo.

assign
/* get the table from the service */
    Tbl = Service:GetTable("Employee")
/* get a new empty policy from the service  */
    policy = Service:NewPartitionPolicy("EmployeePolicy")
    policy:Description = "Employee department partition")
    policy:DefaultDataArea = Service:GetArea("Area1")
    policy:DefaultIndexArea = Service:GetArea("Area1Idx").
    policy:DefaultLobArea = Service:GetArea("Area1Lob")
    policy:DefaultAllocation = "None"
    policy:HasRange = true
    policy:Table = tbl
/* add the field to the policy - found on the table */
policy:Fields:Add(tbl:Fields:Find("DeptCode")).
service:UpdatePolicyDetail(policy).
```

The services `NewPartitionPolicy( )` method expects the name of the policy. The default allocation and default areas are all mandatory properties. The default areas are defined as instances of `IArea( )`, which are retrieved from the service. The `Table` property must also be set before the policy can be submitted to the service. The HasRange and DefaultAllocation properties must also be set before the creation, but these have valid default values and may not need to be set in code.

The example above creates a list partition on the employee table on the department key, and sets the default allocation to `"None"` to allocate the partitions manually later.

Note: You are not allowed to add any data to this table yet since no partition policy details are added to the table.

Editing an existing partition policy

The following code shows how to edit an existing policy:

```
define variable policy as IPartitionPolicy no-undo.
assign
   policy = service:GetPartitionPolicy("OrderlineDatePolicy")
   policy:Description = "Orderlinedate policy").
service:UpdatePartitionPolicy(policy).
```

You can alternatively access the policy from the table:

```
define variable tbl as ITable no-undo.
assign
    table = service:GetTable("Orderline")
    table:PolicyDetail:Description = "Orderlinedate policy").
service:UpdateTable(table).
```

Creating a policy for a table with data

The following code shows how to create a policy for a table that contains data:

```
using OpenEdge.DataAdmin.*.
using OpenEdge.DataAdmin.Error.DataAdminErrorHandler.
using OpenEdge.DataAdmin.Lang.Collections.IIterator.

block-level on error undo, throw.

define variable service as DataAdminService no-undo.

define variable policy as IPartitionPolicy no-undo.
define variable errorHandler as DataAdminErrorHandler no-undo.
define variable tbl  as ITable no-undo.
define variable idx  as IIndex no-undo.

service = new DataAdminService(ldbname(1)).
tbl = service:GetTable("Order").
policy = service:NewPartitionPolicy("OrderPolicy").
policy:Table = tbl.
policy:HasRange = true.

/* create fields from index (can of course just add fields to the Fields collection
directly) */
idx = tbl:Indexes:Find("slsdateidx").
policy:Fields:AddAll(idx:Fields).

policy:DefaultAllocation = "None".
policy:DefaultDataArea  = service:GetArea("TenantExpensive").
policy:DefaultIndexArea  = service:GetArea("TenantExpensive").
policy:DefaultLobArea  = service:GetArea("TenantExpensive").

policy:AddDetailsFromData().

run updateandshowDetails(policy).

service:CreatePartitionPolicy(policy).

catch e as Progress.Lang.Error :
     errorHandler = new DataAdminErrorHandler().
     errorHandler:Error(e).
 end catch.
finally:
     delete object service no-error.
 end finally.

procedure updateAndShowDetails :
    define input  parameter pPolicy as IPartitionPolicy no-undo.
    define variable detail as IPartitionPolicyDetail  no-undo.
    define variable iter as IIterator no-undo.
    iter = pPolicy:Details:Iterator().
    do while iter:HasNext() with frame x down:
        detail = cast(iter:Next(),IPartitionPolicyDetail).
        /* set date to last of year  */
        Detail:SetValue(2,31/12/2014).
        disp detail:Values[1] @ x1 as char
             detail:Values[2] @ x2 as char
               with frame x width 80.
        down with frame x.
    end.
end.
```

The example above runs an internal `updateAndShowDetails` illustrating how you can edit the details before they are committed. The `updateAndShowDetails` changes the `OrderDate` value to the end of the year and displays the two values. The `AddDetailsFromData` creates details with the highest value found in the data.

The following example adds the fields from an index. You can also add the fields directly as follows:

```
...
policy:Fields:Add(tbl:Fields:Find("SalesRep")).
policy:Fields:Add(tbl:Fields:Find("OrderDate")).
...
```

Renaming a partition policy

The following code shows how to rename an existing policy:

```
define variable policy as IPartitionPolicy no-undo.
assign
    policy = service:GetPartitionPolicy("OrderlineDatePolicy")
    policy:Name = "OrderLineDate".
service:UpdatePartitionPolicy(policy).
```

See also:

- IDataAdminElement interface on page 716
- IPartitionMap interface on page 787
- IPartitionPolicy interface on page 733
- IPartitionPolicyDetail interface on page 739
- ITenantGroupSet interface on page 797
- ITenantGroup interface on page 761
- IPartitionMap interface on page 787

IPartitionPolicyDetail interface

The `IPartitionPolicyDetail` interface is an interface that inherits the `IPartitionOwner` interface and is mapped to the `_partition-table` table in the database.

The interface consists of properties that are used to define values for the physical data partition.

The instances are added and accessed through the details collection of the `IPartitionPolicy` interface and are saved to the service with `IPartitionPolicyDetail` interface. The `IPartitionPolicyDetail` interface has no defined methods to read or update policy details.

IPartitionPolicyDetail properties

Inherits properties from `IPartitionOwner` and defines the properties listed in the following table

Table 106: Table Partition Maintenance IPartitionPolicyDetail interface properties

Property name	Data type	Access	Description
`Values`	`CHARACTER`	Read-only	Returns an array with the character representation of the values for a partition policy detail. There is one element for each of the fields of the owning partition policy. The user can edit this only when the policy is new.
`IsDataEnabled`	`LOGICAL`	Readable, Writeable	Set to `TRUE` (the default) allows access to partitioned data. Set to `FALSE` in order to prevent access to partitioned data. This setting leaves only shared data accessible to the partition.Note: Currently, this property returns `FALSE`. Setting this property throws an error.
`IsReadOnly`	`LOGICAL`	Readable, Writeable	Set to TRUE (the default) to mark a partition policy detail as read-only.
`IsSplitTarget`	`Boolean`	Read and write	Specifies whether the partition policy detail is been marked as a split target. That is, the data in the policy detail can be moved to other partitions. For more information on using this property, see PartitionMergeUtility class on page 233.

IPartitionPolicyDetail methods

Inherits methods from `IPartitionOwner` and defines the methods listed in Table 102: Table Partition Maintenance IPartitionParent interface properties on page 732.

Table 107: Table Partition Maintenance IPartitonPolicyDetail interface methods

Method syntax	Description
`VOID SetValue(*)`	Sets the value of a single field partition policy with the native data-type. **Note:** * signifies all Data Types **Note:** The method throws an error if it is called after the policy detail is created, or if it is called in a policy that has multiple fields.
`VOID SetValue(*,INTEGER)`	Sets the value of a multi-field partition policy with the native data-type. **Note:** * signifies all Data Types **Note:** The method throws an error if it is called after the policy detail is created, or if it is called in a policy where the field number (second parameter) is out of bounds of the number of fields in the policy.

IPartitionPolicyDetail examples

The following code illustrates the use of the `IPartitionPolicyDetail` collection for various activities such as finding and updating existing policy details. The `IPartitionPolicyDetail` collection is exposed on the Details property of the parent `IPartitionPolicy`. There is no service API to retrieve or update partition policy details directly:

Creating a new partition detail

The following code shows the creation of a policy with detail values:

```
USING OpenEdge.DataAdmin.Error.DataAdminErrorHandler.
USING OpenEdge.DataAdmin.IPartitionPolicy.
USING OpenEdge.DataAdmin.IPartitionPolicyDetail.
USING OpenEdge.DataAdmin.ITable.
USING OpenEdge.DataAdmin.DataAdminService.

define variable policy as IPartitionPolicy no-undo.
define variable detail as IPartitionPolicyDetail no-undo.
define variable tbl as ITable no-undo.
define variable errorHandler as DataAdminErrorHandler no-undo.
define variable service as DataAdminService no-undo.

service = new DataAdminService(LDBNAME(1)).

assign
    /* get a new empty policy from the service  */
    policy = Service:NewPartitionPolicy("OrderDate")
    policy:Description = "Orderline date partition"
    policy:DefaultDataArea = service:GetArea("Customer/Order Area")
    policy:DefaultIndexArea = service:GetArea("Order Index Area")
    policy:DefaultLobArea = service:GetArea("Customer/Order Area")
    policy:DefaultAllocation = "IMMEDIATE"
    /* get the table from the service */
    Tbl = service:GetTable("Order")
    policy:Table = tbl
    policy:HasRange = YES.

/* add the field to the policy - found on the table */
policy:Fields:Add(tbl:Fields:Find("Order-Date")).

/* get a new empty policy detail from the service  */
detail = Service:NewPartitionPolicyDetail("Data2012").
detail:SetValue(12/31/2012).
policy:Details:Add(detail).

/* get another empty policy detail from the service  */
detail = Service:NewPartitionPolicyDetail("Data2013").
detail:SetValue(12/31/2013).
policy:Details:Add(detail).

service:CreatePartitionPolicy(policy).

catch e as Progress.Lang.Error :
     errorHandler = new DataAdminErrorHandler().
     errorHandler:Error(e).
 end catch.
finally:
     delete object service no-error.
 end finally.
```

The above example partitions the `Order` table on order date, sets the `HasRange` property to TRUE, and adds two ranges so that all data less than the specified values are added to the corresponding partitions. This example assumes there is a table called `Order`, with no records, and an index on a field named `Order-Date`.

Finding by name

The following code shows how to use the `Find(int)` method to locate the instance in the collection using the Partition ID:

```
define variable policy as IPartitionPolicy no-undo.
define variable detail as IPartitionPolicyDetail no-undo.

assign
         policy = service:GetPartitionPolicy("OrderlineDatePolicy")
    detail = policy:Details:Find(23).
```

Finding last by value

The following code illustrates how to use the `FindLast` method to find the partition policy detail that is used to store data of a specific value when the policy is defined as a range. This code checks if all orderlines falls within the defined partitions:

```
define variable policy as IPartitionPolicy no-undo.
define variable detail as IPartitionPolicyDetail no-undo.

policy = service:GetPartitionPolicy("orderlineDate")

for each order no-lock:
    detail = policy:Details:FindLastByValue(order.orderDate).
    if not valid-object(detail) then
    do:
        <do some action for missing partition detail>
    end.
end.
```

Renaming of a partition policy detail

The following code illustrates how to rename a partition policy detail:

```
define variable policy as IPartitionPolicy no-undo.
define variable detail as IPartitionPolicyDetail no-undo.

assign
         policy = service:GetPartitionPolicy("OrderlineDatePolicy")
    detail = policy:Details:Find("OldData")
    detail:Name = "OldDataDetail".

service:UpdatePartitionPolicy(policy).
```

Setting a partition policy detail as read-only

The following code illustrates how to set a partition policy detail read-only:

```
define variable policy as IPartitionPolicy no-undo.
define variable detail as IPartitionPolicyDetail no-undo.

assign
         policy = service:GetPartitionPolicy("OrderlineDate")
    detail = policy:Details:Find("OldData")
    detail:IsReadOnly = true.

service:UpdatePartitionPolicy(policy).
```

See also:

- IDataAdminElement interface on page 716
- IPartitionMap interface on page 787
- IPartitionPolicy interface on page 733
- IPartitionPolicyDetail interface on page 739
- ITenantGroupSet interface on page 797
- ITenantGroup interface on page 761
- IPartitionMap interface on page 787

ISchema interface

The `ISchema` interface is not mapped to any one database table, but represents Data Dictionary schema elements owned by a single schema owner (in `_Owner`). Currently, it supports only schema elements owned by PUB (the OpenEdge default owner). You do not typically need to use this object for regular schema reads and updates because the `Get*( )` and `Update*( )` service methods on `ISequence` and `ITable` also support PUB-owned objects.

The `ISchema` interface also provides support for both read and load of the schema from a `.df` file using dedicated methods of the DataAdminService, including the `GetSchemaChanges( )` and `UpdateSchemaChanges( )` methods (which load the schema into an `ISchema`, then into the database, respectively) and the `LoadSchemaChanges( )` method (which loads the schema into the database in one operation. The benefit of the two-method approach is that the schema has a `LoadOptions` property that allows you to specify options for the load. `ISchema` also has a `Partitions` collection property that holds all new partitions created as a result of the load, allowing you to manage and allocate the new partitions, as needed.

ISchema properties

Inherits properties from `IDataAdminElement` and defines the properties listed in the following table.

Table 108: Multi-tenant Maintenance ISchema interface properties

Property name	Data type	Access	Description
`LoadOptions`	`IDataDefinitionOptions`[16]	Read-only	Provides access to the following IDataDefinitionOptions properties that allow you to read or write the corresponding load options: • **`AddObjectsOnline`** — A `LOGICAL` property that that when set to `TRUE` allows schema to be loaded into an online database, which is the same as setting `SchemaChange` to `"NEW OBJECTS"` • **`FileName`** — A `CHARACTER` property that specifies the name of the data definition (`.df`) file from which schema is loaded • **`ForceCommit`** — A `LOGICAL` property that that when set to `TRUE` forces the changes to the schema to be committed even when an error is raised. • **`ForceIndexDeactivate`** — A `LOGICAL` property that when set to `TRUE` deactivates any new indexes in the loaded schema. • **`ForceSharedSchema`** — A `LOGICAL` property that when set to `TRUE` forces all loaded tables and sequences to be shared, regardless of any multi-tenant settings. • **`SchemaChange`** — A `CHARACTER` property that sets and returns the `SCHEMA-CHANGE` attribute on the `SESSION` system handle (`"NEW OBJECTS"`, blank (`""`), or the Unknown value (?)) All other `IDataDefinitionOptions` properties are reserved for OpenEdge use only.
`Partitions`	`IPartitionCollection`	Read-only	Set to a collection of `IPartition` objects for all partitions in the loaded schema. The value is populated after executing the `UpdateSchemaChanges( )` or `LoadSchemaChanges( )` service methods.

Property name	Data type	Access	Description

ISchema methods

Inherits methods from `IDataAdminElement`.

ISchema examples

The following code allocates new table partitions for the schema changes loaded from the `myschema.df` file:

```
define variable definitions as ISchema   no-undo.
define variable area        as IArea     no-undo.
define variable iterator    as IIterator no-undo.

definitions = service:GetSchemaChanges("myschema.df").
definitions:LoadOptions:ForceIndexDeactivation = true.
service:UpdateSchemaChanges(definitions).
iterator = definitions:Partitions:Iterator("ObjectType = 'table'").
do while iterator:HasNext():
  cast(iterator:Next(),IPartition):Allocate().
end.

service:UpdateSchema(definitions).
```

See also:

- API entities and entity service methods on page 208
- IDataAdminElement interface on page 716
- IPartitionCollection interface on page 785
- Schema (reading, loading, and updating) on page 219

ISchemaElement interface

The `ISchemaElement` is a super interface for the `ITable`, `IIndex`, and `IField` interfaces and allows the `IPartition` interface to have a single property to represent the `ITable`, `IIndex`, or `IField` to which it belongs. The properties and methods of this interface are provided for application use on a sub-interface. Otherwise, the `ISchemaElement` interface is not supported for direct access.

It defines properties common to all schema elements, including the `Partitions` property that can be used to access the schema object's partitions for all tenants and tenant groups.

[16] This interface is currently defined in the OpenEdge.DataAdmin.Binding package, which is supported for OpenEdge use only. You can access the documented properties of the referenced IDataDefinitionOptions object. However, to minimize future changes to your application code, do not define any application data elements using this interface type. A future release of OpenEdge will provide a public interface that you can use to define object references in your code. In this release, make all references to properties of the IDataDefinitionOptions object using chained references directly on the LoadOptions property, for example, LoadOptions:ForceIndexDeactivation = true.

The properties and methods described in the following sections are supported for use in your application code. `ISchemaElement` also defines other properties and methods not described here that are reserved for OpenEdge use only.

ISchemaElement properties

Inherits properties from `IDataAdminElement` and defines the properties listed in the following table.

Table 109: Multi-tenant Maintenance ISchemaElement interface properties

Property name	Data type	Access	Description
`Area`	`IArea`	Read-only	Returns the area (for default tenant if multi-tenant).
`Description`	`CHARACTER`	Read-only	Returns a description of the element.
`Name`	`CHARACTER`	Read-only	Returns the name of the element (table name, index, name or field name).
`ObjectType`	`CHARACTER`	Read-only	Returns object type represented by this object: `"Table"`, `"Index"`, or `"Field"`.
`Partitions`	`ISchemaPartitionMap`	Read-only	Returns the collection of partitions mapped by tenant or tenant group.

ISchemaElement methods

Inherits methods from `IDataAdminElement`.

See also:

- IArea interface on page 710
- IDataAdminElement interface on page 716
- IField interface on page 722
- IIndex interface on page 725
- IPartition interface on page 727
- IPartitionPolicyDetailSet interface on page 789
- ITable interface on page 751

ISequence interface

The ISequence interface the complete maintenance of sequences using the API. The reason sequences are fully supported while schema objects like tables and indexes are not is that an existing sequences cannot become multi-tenant enabled, but must be deleted and replaced with new multi-tenant sequences. Therefore, deleting and creating sequences can be a required task when multi-tenant enabling a database.

ISequence properties

Inherits properties from IDataAdminElement and defines the properties listed in the following table.

Table 110: Multi-tenant Maintenance ISequence interface properties

Property name	Data type	Access	Description
CurrentValue	INT64	Readable, Writeable	Set to the current value of the sequence.
IncrementValue	INT64	Readable, Writeable	Set to the increment value of the sequence.
InitialValue	INT64	Readable, Writeable	Set to the initial value of the sequence.
IsCyclic	LOGICAL	Readable, Writeable	Set to TRUE if the sequence is cyclic (restarts after its most incremented value).
IsMultitenant	LOGICAL	Readable, Writeable	Set to true if the sequence is multi-tenant enabled. Can be writeable only before the sequence is created.
MaximimumValue	INT64	Readable, Writeable	Set to the maximum value of the sequence.
MinimumValue	INT64	Readable, Writeable	Set to the minimum value of the sequence.
Name	CHARACTER	Read-only	Returns the sequence unique name and identifier (in the database). For a new sequence, you specify the name using the DataAdminService NewSequence( ) method.
SequenceValues	ISequenceValueMap	Read-only	Set to a collection of all tenant sequence current values for a multi-tenant sequence. **Restricted access** — Only a super tenant can see sequence values for other tenants.

ISequence methods

Inherits methods from `IDataAdminElement`.

ISequence examples

Creating a sequence

The following code shows assignment of properties for a new sequence:

```
define variable sequence as ISequence no-undo.

assign
     sequence = service:NewSequence("OrderSeq")
  sequence:IsMultitenant = true
  sequence:MaximumValue  = 2000000
  sequence:IncrementValue = 10.

service:CreateSequence(sequence).
```

Setting properties for an existing sequence

The following code edits one of the properties on an existing domain:

```
assign
  sequence = service:GetSequence("OrderSeq").
  sequence:IsCyclic = true.

service:UpdateSequence(sequence).
```

See also:

- IDataAdminElement interface on page 716
- ISequenceSet interface on page 793
- ISequenceValueMap interface on page 794

ISequenceValue interface

An `ISequenceValue` object stores the current value of a multi-tenant sequence for a given regular tenant. Thus, multi-tenant enabled sequence current values are stored per tenant and exposed as a `SequenceValues` property on the `ISequence` and `ITenant` interfaces. This property returns a collection of `ISequenceValue` objects mapped to an `ISequence` or `ITenant` key.

ISequenceValue properties

Inherits properties from `IDataAdminElement` and defines the properties listed in the following table.

Table 111: Multi-tenant Maintenance ISequenceValue interface properties

Property name	Data type	Access	Description
`Sequence`	`ISequence`	Read-only	Returns the sequence with this current value.
`Tenant`	`ITenant`	Read-only	Returns the tenant with this sequence current value.
`CurrentValue`	`INT64`	Readable, Writeable	Set to the sequence current value.

ISequenceValue methods

Inherits methods from `IDataAdminElement`.

ISequenceValue examples

Changing the current value of a sequence for a specific tenant

The following shows how to change the current value of a sequence for a specific tenant:

```
define variable sequence as ISequence no-undo.
define variable sequence as ITenant no-undo.

assign
  sequence = service:GetSequence("OrderSeq")
  tenant   = service:GetTenant("ACME")
  sequence:SequenceValues:Get(tenant):CurrentValue = 2001.

service:UpdateSequence(sequence).
```

Changing the current value of sequence in a tenant's sequence collection

The following shows how to change a tenant's sequence value in the tenant's sequence collection using the sequence as a key:

```
assign
  sequence = service:GetSequence("OrderSeq")
  tenant   = service:GetTenant("ACME")
  tenant:SequenceValues:Get(sequence):CurrentValue = 2001.

service:UpdateTenant(tenant).
```

See also:

- IDataAdminElement interface on page 716
- ISequence interface on page 748
- ISequenceValueMap interface on page 794
- ITenant interface on page 755

ITable interface

The `ITable` interface is mapped to the `_File` table in the schema, and exposes the properties and methods needed for multi-tenancy and table partitioning purposes.

It inherits the `ISchemaElement`, which is also a super interface for the `IIndex` and `IField` interfaces and allows the `IPartition` interface to have a single property that represents the `ITable`, `IIndex`, or `IField` to which it belongs.

You can use the `Partitions` property inherited from `ISchemaElement` to access table partitions for all tenants and tenant groups.

ITable properties

Inherits properties from `ISchemaElement` and defines the properties listed in the following table.

Table 112: Multi-tenant Maintenance and Table Partitioning ITable interface properties

Property name	Data type	Access	Description
`Schema`	`ISchema`	Read-only	Returns the PUB-owned schema for the table.
`IsMultitenant`	`LOGICAL`	Readable, Writeable	Set to TRUE if want the table to be multi-tenant. This property is writeable only when it is set to FALSE or when it is set to TRUE and not yet committed to the database. Once you make a table in the database multi-tenant you cannot make it shared again. However, you can create a tenant group for the table that is shared by any or all regular tenants in the database that you specify. For more information, see ITenantGroup interface on page 761.
`KeepDefaultArea`	`LOGICAL`	Read-only	Returns TRUE if the table has a default [tenant] data area.
`Indexes`	`IIndexSet`	Read-only	Returns a collection of the table's indexes.

Property name	Data type	Access	Description
`Fields`	`IFieldSet`	Read-only	Returns a collection of the table's fields.
`IsPartitioned`	`LOGICAL`	Read-only	Returns TRUE if the table is partitioned and has a partition policy.
`PartitionPolicy`	`IPARTITIONPOLICY`	??	Returns the partition policy of the table.

ITable methods

Inherits methods from `ISchemaElement` and defines the methods listed in the following table.

Table 113: Multi-tenant Maintenance ITable interface methods

Method syntax	Description
`LOGICAL Allocate ( )`	Allocates all delayed table, field, and index partitions for all tenants. Does not allocate partitions for tenant groups. Returns TRUE if any partition was allocated.
`LOGICAL Allocate (` `  pcAllocateState AS CHARACTER )`	Allocates tenant partitions for the table, its fields, and its indexes depending on the partition state specified by *`pcAllocateState`*, which can be `"All"`, `"Delayed"`, or `"None"`. Does not allocate partitions for tenant groups. Returns TRUE if any partition was allocated.

ITable examples

The following examples show different ways to allocate table partitions accessed from an `ITable` object.

Allocating table partitions

The `ITable` interface has an `Allocate( )` method that allocates partitions for all delayed table, index, and LOB field partitions for the specified table. (The method is also overloaded with a parameter that can specify whether to allocate partitions in the `"All"`, `"Delayed"`, or `"None"` state):

```
assign
  service   = new DataAdminService("sportsmt")
  tableImpl = service:GetTable("Customer")
  tableImpl:Allocate( ).

service:UpdateTable(tableImpl).
```

Note: The `Allocate( )` method does not allocate partitions that belong to a tenant group defined for the table. The `ITenantGroup` interface has its own `Allocate( )` method.

Managing partitions for a table

The following code enables a table for multi-tenancy and changes the area for its index, field, and table partitions. It also uses the `Allocate( )` methods on the table partitions, which also allocate the index and field partitions of the table. (The `Allocate( )` method cannot be used directly on index and field partitions):

```
using OpenEdge.DataAdmin.*.
using OpenEdge.DataAdmin.Lang.Collections.*.

define variable service  as DataAdminService no-undo.
define variable tableImpl  as ITable no-undo.
define variable iterator as IIterator no-undo.
define variable iterator2 as IIterator no-undo.
define variable partition as IPartition no-undo.

assign
  service = new DataAdminService("sportsmt")
  tableImpl = service:GetTable("Customer")
  tableImpl:IsMultitenant = true.

service:UpdateTable(tableImpl).
iterator = TableImpl:LOBFields:Iterator().
do while iterator:HasNext():
  iterator2 = cast(iterator:Next(),IField):Partitions:Iterator("AllocationState <>
'Allocated'").
  do while iterator2:HasNext():
    partition = cast(iterator2:Next(),IPartition).
    partition:Area = service:GetArea("ExpressLobArea").
  end.
end.

iterator = TableImpl:Indexes:Iterator().
do while iterator:HasNext():
  iterator2 = cast(iterator:Next(),IIndex):Partitions:Iterator("AllocationState <>
'Allocated'").
  do while iterator2:HasNext():
    partition = cast(iterator2:Next(),IPartition).
    partition:Area = service:GetArea("ExpressIndexArea").
  end.
end.

iterator = tableImpl:Partitions:Iterator("AllocationState <> 'Allocated'").
do while iterator:HasNext():
  partition = cast(iterator:Next(),IPartition).
  partition:Area = service:GetArea("ExpressDataArea").
  partition:Allocate().
end.
service:UpdateTable(tableImpl).
```

Note: The invocation of `partition:Allocate( )`is specified only for table partitions (in the final `DO` loop), because all LOB field and index partitions are allocated automatically along with their parent table partitions.

See also:

- IField interface on page 722
- IFieldSet interface on page 782
- IIndex interface on page 725
- IIndexSet interface on page 783

- IPartition interface on page 727
- IPartitionOwner interface on page 729
- ISchemaElement interface on page 746
- ITableSet interface on page 796

ITablePermission interface

The `ITablePermission` interface is mapped to the `_File` table and exposes the security related fields as properties.

ITablePermission properties

Inherits properties from `IDataAdminElement` and defines the properties listed in the following table.

Table 114: Multi-tenant Maintenance ITablePermission interface properties

Property name	Data type	Access	Description
`CanCreate`	`CHARACTER`	Readable, Writeable	Set to a `CAN-DO` user ID permissions expression that defines who can create table records.[17]
`CanDelete`	`CHARACTER`	Readable, Writeable	Set to a `CAN-DO` user ID permissions expression that defines who can delete table records.[17]
`CanDump`	`CHARACTER`	Readable, Writeable	Set to a `CAN-DO` user ID permissions expression that defines who can dump table records.[17]
`CanLoad`	`CHARACTER`	Readable, Writeable	Set to a `CAN-DO` user ID permissions expression that defines who can load table records.[17]
`CanRead`	`CHARACTER`	Readable, Writeable	Set to a `CAN-DO` user ID permissions expression that defines who can read table records.[17]
`CanWrite`	`CHARACTER`	Readable, Writeable	Set to a `CAN-DO` user ID permissions expression that defines who can write to table records.[17]
`FieldPermissions`	`IFieldPermissionSet`	Read-only	Returns a collection of field permissions for all the fields in the table.

[17] For more information, see the `CAN-DO` function entry in *ABL Reference* and the sections on user ID patterns and ABL permissions checking in *Introduction to Identity Management*.

Property name	Data type	Access	Description
Frozen	LOGICAL	Read-only	Returns TRUE if the table is frozen.[18]
Hidden	LOGICAL	Read-only	Returns TRUE if the table is hidden.[18]
Name	CHARACTER	Read-only	Returns the table name.
Schema	ISchema	Read-only	Returns the table schema.

ITablePermission methods

Inherits methods from IDataAdminElement.

ITablePermission examples

The following code retrieves an ITablePermission object from the service, edits the properties, and passes the object to the service to be committed to the database:

```
define variable tblPerm as ITablePermission no-undo.
define variable domain as IDomain no-undo.

tblPerm = service:GetTablePermission("Customer").
tblPerm:CanCreate =  "xx@ABCDomain".
tblPerm:CanWrite =  "xx@ABCDomain".
service:UpdateTablePermission(tblPerm).
```

See also:

- IDataAdminElement interface on page 716
- IFieldPermissionSet interface on page 782
- IPartitionOwner interface on page 729
- ITablePermissionSet interface on page 795

ITenant interface

The ITenant interface provides the properties and methods needed to work with tenants. It is mapped to the _Tenant table in the database. In addition to regular data properties, ITenant provides Domains, Partitions, TenantGroups, and SequenceValues as properties that return a collections of related entities.

[18] The Frozen and Hidden properties are read-only while the _Can* fields that define who has access permissions to the table are writeable.

ITenant properties

Inherits properties from `IDataAdminElement` and defines the properties listed in the following table.

Table 115: Multi-tenant Maintenance ITenant interface properties

Property name	Data type	Access	Description
`DefaultAllocation`	`CHARACTER`	Readable, Writeable	Set to a preference that decides whether to allocate partitions when the tenant is created, when tables are multi-tenant enabled, or new multi-tenant tables, indexes, or LOB fields are created. This preference can have the following values: • **`"Immediate"`** — Allocate immediately using default areas. Allocates all partitions according to the defaults when the tenant is created. • **`"Delayed"`** — Delay the physical allocation of the partitions. Sets the partitions' `AllocateState` properties to `"Delayed"`, which allows them to be allocated by executing the `Allocate( )` method on the `ITenant` object. • **`"None"`** — No automatic allocation. Sets the partitions' `AllocateState` properties to `"None"`, which means that the allocation can only be done on each individual partition by executing the `Allocate( )` method on the `IPartition` object. **Note:** If you intend to make a tenant a member of a tenant group, set this property to `"Delayed"` so no storage is allocated to the tenant before you have allocated storage to the group.
`DefaultDataArea`	`IArea`	Readable, Writeable	Set to the default data area for new tenant partitions.
`DefaultIndexArea`	`IArea`	Readable, Writeable	Set to the default index area for new tenant partitions.
`DefaultLobArea`	`IArea`	Readable, Writeable	Set to the default LOB area for new tenant partitions.

Property name	Data type	Access	Description
Description	CHARACTER	Readable, Writeable	Set to a tenant description of your choice.
Domains	IDomainSet	Read-only	Returns a collection of all domains (IDomain) configured for the tenant.
ExternalId	CHARACTER	Readable, Writeable	Set to an external tenant identifier of your choice, with no requirement for a GUID.
Id	INTEGER	Read-only	Returns the unique, internal tenant identifier that is generated by the database. **Note:** Dump and load generates a new value.
IsAllocated	LOGICAL	Read-only	Returns TRUE if all tenant partitions have a state of "Allocated" or "None".Returns FALSE if any tenant partition has a state of "Delayed". **Note:** Use the Allocate( ) method on ITenant to allocate delayed partitions for a tenant.
IsDataEnabled	LOGICAL	Readable, Writeable	Set to TRUE (the default) allows access to tenant data.Set to FALSE in order to prevent access to tenant data. This setting leaves only shared data accessible to the tenant.
Name	CHARACTER	Readable, Writeable	Set to a unique tenant name, with a maximum length 32 characters.
Partitions	IPartitionMap	Read-only	Returns a collection of all partitions of the tenant mapped by ISchemaElement (ITable, IIndex, and IField).
SequenceValues	ISequenceValueMap	Read-only	Returns a collection of multi-tenant sequence values, consisting of ISequenceValue elements mapped by ISequence elements for the tenant.
TenantGroups	ITenantGroupSet	Read-only	Returns a collection all tenant groups (ITenantGroup) to which the tenant belongs.

Property name	Data type	Access	Description
Type	CHARACTER	Readable, Writeable	Set to the tenant type, which can have the following valid values: "Regular" (for a regular tenant) and "Super" (for a super tenant).
Users	IUserSet	Read-only	Returns a collection of all OpenEdge users (IUser) defined for the tenant. **Note:** You can also access OpenEdge tenant users through the IDomain objects returned by the Domains property.

ITenant methods

Inherits methods from IDataAdminElement and defines the methods listed in the following table.

Table 116: Multi-tenant Maintenance ITenant interface methods

Method syntax	Description
LOGICAL Allocate ( )	Allocates all the tenant's partitions. **Note:** The tenant (ITenant) must be passed to a tenant Update*( ) method on the service in order to commit the changes.

ITenant examples

The following sections show examples for a variety of coding tasks when working with tenants.

Creating and editing a tenant

The following code shows assignment of properties and creation of a new tenant. The service's `NewTenant( )` factory method expects the name of the tenant. The `Type`, `DefaultAllocation`, and each of the `Default*Area` properties are all mandatory properties that need to be specified before the tenant can be passed to the `CreateTenantGroup( )` service method. The `Default*Area` properties are all defined as instances of `IArea`, which must be known to, and can be retrieved from, the service. This example sets the default allocation to `"Immediate"` in order to allocate the partitions according to the default areas when the tenant is created:

```
define variable tenant as ITenant no-undo.

assign
  tenant = service:NewTenant("Acme")
  tenant:Type = "Regular"
  tenant:DefaultAllocation = "Immediate"
  tenant:Description = "This company is an important player in its segment.")
  tenant:DefaultDataArea  = service:GetArea("FirstClassDataArea")
  tenant:DefaultIndexArea = service:GetArea("FirstClassIndexArea")
  tenant:DefaultLobArea   = service:GetArea("GeneralLobArea").

service:CreateTenant(tenant).
```

All writeable properties except `Type` can be edited after the tenant is created. The following code edits two of the properties on an existing tenant. Note that `IsDataEnabled` defaults to `TRUE` when the tenant is created:

```
assign
  tenant = service:GetTenant("Acme")
  tenant:Description = tenant:Description + " But is not ready to go online")
  tenant: IsDataEnabled = false.
service:UpdateTenant(tenant).
```

Note: There must be at least one domain configured for a tenant with at least one user in order to login and access tenant data. The next section discusses how to add domains to the tenant.

Adding domains to a tenant

The following code shows how to use the `Domains` property to add a new domain to an existing tenant:

```
define variable tenant as ITenant no-undo.
define variable domain as IDomain no-undo.

tenant = service:GetTenant("ABC")
domain = service:NewDomain("ABCDomain").
domain:Description = "Domain for the ABC company").

tenant:Domains:Add(domain).

service:UpdateTenant(tenant).
```

A tenant must have at least one domain configured for it before a tenant user can access the data. The `Tenant` property on `IDomain` is updated automatically when it is added to `Domains` property collection of an `ITenant`. For more information, see IDomain interface on page 718.

Note: There must be at least one user in the tenant domain in order to login and access tenant data. This user must have an account in the authentication system configured for the domain. For more information on adding OpenEdge `_User` table users to a domain, see IDomain examples on page 720.

Adding tenant groups to a tenant

You can assign a tenant to one or more tenant groups by adding tenant groups to an `ITenant`. The following code shows how you can use the `TenantGroups` property to add tenant groups to the tenant:

```
define variable tenant as ITenant no-undo.
define variable tenantgroup as ITenantGroup no-undo.

 assign
  tenant = service:GetTenant("ACME")
  tenantgroup = service:GetTenantGroup("AcmeCustomer")
  tenant:TenantGroups:Add(tenantgroup)
  tenantgroup = service:GetTenantGroup("AcmeOrder")
  tenant:TenantGroups:Add(tenantgroup).

service:UpdateTenantGroup(tenantgroup).
```

Note: Any `ITenant` that you intend to make a member of a group should be created with its `DefaultAllocation` property set to `"Delayed"` so tenant storage is not allocated in addition to the storage for the group.

Managing partitions for a tenant

Tenant partitioning is managed by various default settings on the tenant. The partitions for a tenant can be accessed through the `Partitions` property collection on the `ITenant` interface.

Setting `DefaultAllocation` to `"Immediate"` causes the partitioning to take place in the same transaction as the tenant creation using the default areas defined for the tenant. Setting the property to `"Delayed"` or `"None"` postpones the allocation and allows the area to be edited before the allocation.

The `"Delayed"` setting allows the partitions to be allocated in one operation by invoking the `Allocate( )` method on `ITenant`, which sets the `AllocationState` property for every tenant partition (`IPartition`) to `"Allocated"`. However, the `IsAllocated` property on `ITenant` returns FALSE when any individual partition has an `AllocationState` setting of `"Delayed"`. The `"None"` setting is considered a permanent setting, and partitions that have an `AllocationState` setting of `"None"` do not affect a tenant's `IsAllocated` property setting. Also, allocating individual partitions do not affect the tenant's `IsAllocated` property setting. So, while changing the `AllocationState` setting of a partition from `"None"` to `"Delayed"` sets the tenant's `IsAllocated` property to FALSE, changing the `AllocationState` setting of a partition to `"Allocated"` does not set the tenant's `IsAllocated` property to TRUE.

Note: The `IsAllocated` property on `ITenant` does not necessarily reflect the state of all the tenant partitions. While it is always FALSE if one of the tenant partitions has an `AllocationState` setting of `"Delayed"`, `IsAllocated` can be set to TRUE only by invoking the `Allocate( )` method on `ITenant`. It **cannot** be set to TRUE by invoking the `Allocate( )` method on the `Partitions` property (an `IPartitionMap`) of `ITenant` or on any individual partition (`IPartition`). In other words, for `IsAllocated` to be TRUE for a tenant, the `Allocate( )` method must be invoked on the tenant itself, **not** on any one or more of the partitions (in a collection) owned by the tenant.

The following example creates the tenant with a delayed default allocation and changes the area on two partitions before the allocation is done:

```
define variable tenant   as ITenant no-undo.
define variable tbl       as ITable no-undo.
define variable partition as IPartition no-undo.

assign
  tenant = service:NewTenant("ABC")
  tenant:DefaultDataArea  = service:GetArea("FirstClassDataArea")
  tenant:DefaultIndexArea = service:GetArea("FirstClassIndexArea")
  tenant:DefaultLobArea   = service:GetArea("GeneralLobArea").
  tenant:DefaultAllocation = "Delayed".

service:CreateTenant(tenant).

assign
  tbl = service:GetTable("Customer")
  partition = tenant:Partitions:Get(tbl)
  partition:Area = service:GetArea("CustomerArea")
  partition = tenant:Partitions:Get(tbl:Indexes:Find("CustIdx"))
  partition:Area = service:GetArea("CustIndexArea").

tenant:Allocate().
Service:UpdateTenant(tenant).
```

See also:

- IArea interface on page 710
- IDataAdminElement interface on page 716
- IDomain interface on page 718
- IDomainSet interface on page 780
- IPartition interface on page 727
- IPartitionMap interface on page 787
- ISequence interface on page 748
- ISequenceValue interface on page 749
- ISequenceValueMap interface on page 794
- ITenantGroup interface on page 761
- ITenantGroupSet interface on page 797
- ITenantSet interface on page 798
- IUser interface on page 766
- IUserSet interface on page 801

ITenantGroup interface

The `ITenantGroup` interface provides the properties and methods required to work with tenant groups. `ITenantGroup` is mapped to the `_Partition-Set` table in the database.

In addition to regular data properties, ITenantGroup provides Tenants and Partitions properties that returns a collection of the tenants assigned to the group and the partitions of the group, respectively.

ITenantGroup properties

Inherits properties from IDataAdminElement and defines the properties listed in the following table.

Table 117: Multi-tenant Maintenance ITenantGroup interface properties

Property name	Data type	Access	Description
DefaultAllocation	CHARACTER	Readable, Writeable	Set to a preference that decides whether to allocate partitions when the tenant group is created, or when new multi-tenant indexes or LOB fields are created on the table of the group. This preference can have the following values: • **"Immediate"** — Allocate immediately using default areas. Allocates all partitions according to the defaults when the tenant group is created. • **"Delayed"** — Delay the physical allocation of the partitions. Sets the partitions' AllocateState properties to "Delayed", which allows them to be allocated by executing the Allocate( ) method on the ITenantGroup object. • **"None"** — No automatic allocation. Sets the partitions' AllocateState properties to "None", which means that the allocation can only be done on each individual partition by executing the Allocate( ) method on the IPartition object.
DefaultDataArea	IArea	Readable, Writeable	Set to the default data area for new tenant group partitions.
DefaultIndexArea	IArea	Readable, Writeable	Set to the default index area for new tenant group partitions.
DefaultLobArea	IArea	Readable, Writeable	Set to the default LOB area for new tenant group partitions.
Description	CHARACTER	Readable, Writeable	Set to a tenant group description of your choice.

Property name	Data type	Access	Description
Id	INTEGER	Read-only	Returns the unique, internal tenant group identifier that is generated by the database. **Note:** Dump and load generates a new value.
IsAllocated	LOGICAL	Read-only	Returns TRUE if all tenant group partitions have a state of "Allocated" or "None". Returns FALSE if any partition has a state of "Delayed". **Note:** Use the Allocate( ) method to allocate delayed partitions.
Name	CHARACTER	Readable, Writeable	Set to a unique tenant group name, with a maximum length 32 characters.
Partitions	IPartitionMap	Read-only	Returns a collection of all partitions of the tenant group mapped by ISchemaElement (ITable, IIndex, and IField).
Table	ITable	Read-only	Set to the table of the group.
Tenants	ITenantSet	Read-only	Returns a collection all tenants (ITenant) that are members of the group.

ITenantGroup methods

Inherits methods from IDataAdminElement and defines the methods listed in the following table.

Table 118: Multi-tenant Maintenance ITenantGroup interface methods

Method syntax	Description
LOGICAL Allocate ( )	Allocates all the tenant group's partitions. **Note:** The tenant group (ITenantGroup) must be passed to a tenant group Update*( ) method on the service in order to commit the changes.

ITenantGroup examples

The following sections show examples for a variety of coding tasks when working with tenant groups.

Creating and editing a tenant group

The following code shows assignment of properties and creation of a new tenant group. The service's `NewTenantGroup( )` factory method expects the name of the group. The `Table`, `DefaultAllocation`, and each of the `Default*Area` properties are all mandatory properties that need to be specified before the tenant group can be passed to the `CreateTenantGroup( )` service method. The `Default*Area` properties are all defined as instances of `IArea`, which must be known to, and can be retrieved from, the service. This example sets the default allocation to `"Immediate"` in order to allocate the partitions according to the default areas when the tenant group is created:

```
define variable tenantgroup as ITenantGroup no-undo.

assign
  tenantgroup = service:NewTenantGroup("AcmeCustomer")
  tenantgroup:Table = service:GetTable("Customer")
  tenantgroup:DefaultAllocation = "Immediate"
  tenantgroup:Description = "This group is for the customer table for ACME.")
  tenantgroup:DefaultDataArea  = service:GetArea("GeneralDataArea")
  tenantgroup:DefaultIndexArea = service:GetArea("FirstClassIndexArea")
  tenantgroup:DefaultLobArea   = service:GetArea("GeneralLobArea").

service:CreateTenantGroup(tenantgroup).
```

All writeable properties except `Table` can be edited after the tenant group is created. The following code changes the `DefaultDataArea` setting and sets `DefaultAllocation` to `"None"` in order to be able to allocate partitions manually:

```
assign
  tenantgroup = service:GetTenantGroup("AcmeCustomer")
  tenantgroup:DefaultDataArea = service:GetArea("FirstClassDataArea")
  tenantgroup:DefaultAllocation = "None".

service:UpdateTenantGroup(tenantgroup).
```

Adding tenants to a tenant group

You can assign one or more tenants to a tenant group. The following code shows how you can use the `Tenants` property to add tenants to a group:

```
define variable tenantgroup as ITenantGroup no-undo.

 assign
  tenantgroup = service:GetTenantGroup("AcmeCustomer")
  tenantgroup:Tenants:Add(service:GetTenant("ACME-1"))
  tenantgroup:Tenants:Add(service:GetTenant("ACME-2")).

service:UpdateTenantGroup(tenantgroup).
```

Managing partitions for a tenant group

Tenant group partitioning is managed by various default settings on the tenant group. The partitions for a tenant group can be accessed through the `Partitions` property collection on the `ITenantGroup` interface.

Setting `DefaultAllocation` to `"Immediate"` causes the partitioning to take place in the same transaction as the tenant group creation using the default areas defined for the tenant group. Setting the property to `"Delayed"` or `"None"` postpones the allocation and allows the area to be edited before the allocation.

The `"Delayed"` setting allows the partitions to be allocated in one operation by invoking the `Allocate( )` method on `ITenantGroup`, which sets the `AllocationState` property for every tenant group partition (`IPartition`) to `"Allocated"`. However, the `IsAllocated` property on `ITenantGroup` returns FALSE when any individual partition has an `AllocationState` setting of `"Delayed"`. The `"None"` setting is considered a permanent setting, and partitions that have an `AllocationState` setting of `"None"` do not affect a tenant group's `IsAllocated` property setting. Also, allocating individual partitions do not affect the tenant group's `IsAllocated` property setting. So, while changing the `AllocationState` setting of a partition from `"None"` to `"Delayed"` sets the tenant group's `IsAllocated` property to FALSE, changing the `AllocationState` setting of an individual partition to `"Allocated"` does not set the tenant group's `IsAllocated` property to TRUE.

Note: The `IsAllocated` property on `ITenantGroup` does not necessarily reflect the state of all the tenant group partitions. While it is always FALSE if one of the tenant group partitions has an `AllocationState` setting of `"Delayed"`, `IsAllocated` can be set to TRUE only by invoking the `Allocate( )` method on `ITenantGroup`. It **cannot** be set to TRUE by invoking the `Allocate( )` method on the `Partitions` property (an `IPartitionMap`) of `ITenantGroup` or on any individual partition (`IPartition`). In other words, for `IsAllocated` to be TRUE for a tenant group, the `Allocate( )` method must be invoked on the tenant group itself, **not** on any one or more of the partitions (in a collection) owned by the tenant group.

The following example creates a tenant group with `DefaultAllocation` set to `"None"` to postpone the allocation and changes the area on one index partition before allocating all partitions:

```
define variable tenantgroup as ITenantGroup no-undo.
define variable partition as IPartition no-undo.

assign
  tenantgroup = service:NewTenantGroup ("ABCEmployee")
  tenantgroup:Table      = service:GetTable("Employee")
  tenantgroup:DefaultDataArea  = service:GetArea("FirstClassDataArea")
  tenantgroup:DefaultIndexArea = service:GetArea("FirstClassIndexArea")
  tenantgroup:DefaultLobArea   = service:GetArea("GeneralLobArea").
  tenantgroup:DefaultAllocation = "None".

service:CreateTenantGroup(tenantgroup).

assign
  partition = tenantgroup:Partitions:Get(
              tenantgroup:Table:Indexes("EmployeeIdx"))
  partition:Area = service:GetArea("EmpIndexArea").

tenantgroup:Allocate().
Service:UpdateTenantGroup(tenantgroup).
```

See also:

- IArea interface on page 710
- IDataAdminElement interface on page 716

- ITable interface on page 751
- ITenant interface on page 755
- ITenantGroupSet interface on page 797
- ITenantSet interface on page 798
- IPartition interface on page 727
- IPartitionMap interface on page 787

IUser interface

The `IUser` interface provides properties to work with users whose accounts are stored in the `_User` database table. For more information on adding OpenEdge `_User` table users to a domain, see IDomain examples on page 720.

IUser properties

Inherits properties from `IDataAdminElement` and defines the properties listed in the following table.

Table 119: Multi-tenant Maintenance IUser interface properties

Property name	Data type	Access	Description
`Id`	`CHARACTER`	Read-only	Returns the unique user ID (*`username@domainname`*).
`Name`	`CHARACTER`	Readable, Writeable	Set to the unique name of the user within the domain. Provides the *`username`* part of the user ID (*`username@domainname`*).
`Domain`	`IDomain`	Readable, Writeable	Set to the domain of the user. The `Name` property on `IDomain` provides the *`domainname`* part of the user ID (*`username@domainname`*).
`Tenant`	`ITenant`	Read-only	Returns the tenant of the user as defined by the `Domain` property.
`Description`	`CHARACTER`	Readable, Writeable	Set to an optional free form text description of the user.
`IsSqlOnly`	`LOGICAL`	Readable, Writeable	Set to `TRUE` if the user is an SQL-only user (ignored by the AVM).
`GivenName`	`CHARACTER`	Readable, Writeable	Set to the given name of the user.
`MiddleInitial`	`CHARACTER`	Readable, Writeable	Set to the middle initial of the user.

Property name	Data type	Access	Description
SurName	CHARACTER	Readable, Writeable	Set to the surname (family or last name) of the user.
Password	CHARACTER	Readable, Writeable	Set to the user's password. This is writeable only by the user.
Telephone	CHARACTER	Readable, Writeable	Set to the optional telephone number of the user.
Email	CHARACTER	Readable, Writeable	Set to the optional e-mail address of the user.

IUser methods

Inherits methods from IDataAdminElement.

IUser examples

Creating and editing a user

The following code shows assignment of properties for a new user:

```
define variable user as IUser no-undo.
assign
  user = service:NewUser("mcblack")
  user:Password = /* application variable */
  user:GivenName = "Martin"
  user:SurName  = "Black"
  user:Domain = service:GetDomain("AcmeDomain").

service:CreateUser(user).
```

After the user has been created the password can only be changed by the actual user.

Editing an existing user

The following code edits a property on an existing user:

```
assign
  user = service:GetUser("mblack@AcmeDomain")
  user:Telephone = "5896001222".

service:UpdateUser(user).
```

See also:

- IDataAdminElement interface on page 716

- IDomain interface on page 718
- ITenant interface on page 755
- IUserSet interface on page 801

IUserFieldPermission interface

The `IUserFieldPermission` interface represents the field permissions for a user and a field. The user is defined by an `IUserPermission` object and can, but does not need to be defined in the `_User` table in the metaschema. This allows the interface to be used to check permissions for users defined in authentication systems other than `_oeusertable`.

All the properties in this interface are read-only.

IUserFieldPermission properties

Inherits properties from `IDataAdminElement` and defines the properties listed in the following table.

Table 120: Multi-tenant Maintenance IUserFieldPermission interface properties

Property name	Data type	Access	Description
`CanRead`	`LOGICAL`	Read-only	Returns `TRUE` if the user specified by an owning `IUserPermission` object can read the field.
`CanWrite`	`LOGICAL`	Read-only	Returns `TRUE` if the user specified by an owning `IUserPermission` object can write the field.
`Name`	`CHARACTER`	Read-only	Returns the name of a field in a table specified by an owning `IUserTablePermission` object.

IUserFieldPermission methods

Inherits methods from `IDataAdminElement`.

See also:

- IDataAdminElement interface on page 716
- IUserFieldPermissionSet interface on page 799
- IUserPermission interface on page 769
- IUserTablePermission interface on page 770

IUserPermission interface

The IUserPermission interface represents a user and has an IUserTablePermissionSet object with the user permissions for each table in the database. The user can be defined in the _User table in the metaschema, but the IUserPermission object can also be used to retrieve permission information for users that are not defined in the database.

All the properties in this interface are read-only.

IUserPermission properties

Inherits properties from IDataAdminElement and defines the properties listed in the following table.

Table 121: Multi-tenant Maintenance IUserPermission interface properties

Property name	Data type	Access	Description
Domain	IDomain	Read-only	Returns the domain of the user. The Name property on IDomain provides the *domainname* part of the user ID (*username@domainname*).
Id	CHARACTER	Read-only	Returns the unique user ID (*username@domainname*).
Name	CHARACTER	Read-only	Returns the unique name of the user within the domain. Provides the *username* part of the user ID (*username@domainname*).
UserTablePermissions	IUserTablePermissionSet	Read-only	Returns a collection of IUserTablePermission objects that define the user permissions for all the tables and fields in the database.

IUserPermission methods

Inherits methods from IDataAdminElement.

See also:

- IDataAdminElement interface on page 716
- IUserPermissionSet interface on page 800
- IUserTablePermission interface on page 770
- IUserTablePermissionSet interface on page 802

IUserTablePermission interface

The IUserTablePermission interface represents the permissions for a user and a specific table. The user is defined by an IUserPermission object and can, but does not need to be defined in a _User record of the metaschema. This allows the interface to be used to check permission for users defined in authentication systems other than _oeusertable.

All the properties in this interface are read-only.

The UserFieldPermissions property returns a collection of IUserFieldPermission instances for all fields of the table. A set of LOGICAL Can* properties return TRUE or FALSE indicating whether the user can perform the specified operation on a table. The remaining properties identify the table and its status in the database.

IUserTablePermission properties

Inherits properties from IDataAdminElement and defines the properties listed in the following table.

Table 122: Multi-tenant Maintenance IUserTablePermission interface properties

Property name	Data type	Access	Description
CanCreate	LOGICAL	Read-only	Returns TRUE if the user specified by an owning IUserPermission object can create records in this table.
CanDelete	LOGICAL	Read-only	Returns TRUE if the user specified by an owning IUserPermission object can delete records in this table.
CanDump	LOGICAL	Read-only	Returns TRUE if the user specified by an owning IUserPermission object can dump records in this table.
CanLoad	LOGICAL	Read-only	Returns TRUE if the user specified by an owning IUserPermission object can load records in this table.
CanRead	LOGICAL	Read-only	Returns TRUE if the user specified by an owning IUserPermission object can read records in this table.
CanWrite	LOGICAL	Read-only	Returns TRUE if the user specified by an owning IUserPermission object can write to records in this table.
Frozen	LOGICAL	Read-only	Returns TRUE if the this table is frozen in the database.
Hidden	LOGICAL	Read-only	Returns TRUE if the this table is hidden in the database.

Property name	Data type	Access	Description
`Name`	`CHARACTER`	Read-only	Returns the name of the table in the database
`UserFieldPermissions`	`IUserFieldPermissionSet`	Read-only	Returns a collection of `IUserFieldPermission` objects that identify the user permissions for all the fields in the table.

IUserTablePermission methods

Inherits methods from `IDataAdminElement`.

See also:

- IDataAdminElement interface on page 716
- IUserPermission interface on page 769
- IUserFieldPermission interface on page 768
- IUserFieldPermissionSet interface on page 799
- IUserTablePermissionSet interface on page 802

G

Database Administration Entity Collection Interface Reference

This appendix provides a reference to entity collection interfaces that support the ABL API for Multi-tenant Maintenance. For more information on this API, see ABL API for Multi-tenant and Table Partition Management on page 205. These entity collection interfaces provide public access to classes that implement specialized collections of most of the various sub-interface objects of `IDataAdminElement`, which is the super interface of all entity interfaces in the API. Many of the collections can be retrieved and updated directly using the DataAdminService. The collections are also used to represent the "many" end of a one-to-many or many-to-many relationship among the various entity objects, in which case the service can read and update them together with their parent.

All collection interfaces (and classes) inherit (or implement) the `IDataAdminCollection` interface, and in some cases, other interfaces. In the sections that describe these interfaces, the interface names are referenced without their containing packages, which include:

- **`OpenEdge.DataAdmin`** — For both entity collection interfaces and the entity interfaces of objects stored in these API collections (documented in Database Administration Entity Interface Reference on page 707)
- **`OpenEdge.DataAdmin.Lang.Collections`** — For the general iterator interface, `IIterator`, used to navigate the entity objects in a collection.

Any code s in this appendix assume required `USING` statement declarations for `OpenEdge.DataAdmin.*` and `OpenEdge.DataAdmin.Lang.Collections.*`, and the definition and instantiation of the DataAdminService (`service`) as shown in Instantiating a DataAdminService on page 213.

For details, see the following topics:

- Collection interface naming conventions
- Common collection features

- IAreaSet interface
- IAuthenticationSystemSet interface
- IDataAdminCollection interface
- IDomainSet interface
- IExtentList interface
- IFieldPermissionSet interface
- IFieldSet interface
- IIndexSet interface
- IIterator interface
- IPartitionCollection interface
- IPartitionMap interface
- IPartitionPolicyDetailSet interface
- IPartitionPolicySet interface
- ISchemaPartitionMap interface
- ISequenceSet interface
- ISequenceValueMap interface
- ITablePermissionSet interface
- ITableSet interface
- ITenantGroupSet interface
- ITenantSet interface
- IUserFieldPermissionSet interface
- IUserPermissionSet interface
- IUserSet interface
- IUserTablePermissionSet interface

Collection interface naming conventions

In general, the name of each collection interface consists of the following components:

Syntax

```
{I}{Entity}{CollectionType}
```

`I`

Identifies the name of an interface.

`Entity`

The name of the entity that the interface represents, for instance, `Domain` or `Tenant`.

`CollectionType`

The type of collection, which can contain multiple instances of *`Entity`*. This can be one of the following:

- `Collection` — Used for a few base interfaces and classes, for instance, `IDataAdminCollection`, which is inherited by all collection interfaces.
- `List` — Used for collections that are an ordered collection (sequence), for instance, `IExtentList` (currently the only list).
- `Map` — Used for collections that map entity values to keys (with no duplicate keys), for instance, `IPartitionMap` and `ISequenceValueMap`. In effect, entity maps relate collections of entities in many-to-many relationships, such as a collection of multi-tenant sequences mapped to a collection of tenants through the sequence values (the `ISequenceValueMap`).

  Note: A `Map` is not used to map scalar keys of type `INTEGER` or `CHARACTER`. Instead, collections can have `Find( )` and `CanFind( )` methods for scalar properties (keys).

- `Set` — Used for a collection of entities that contain no duplicate elements.

Common collection features

All the collection implementations override the `Progress.Lang.Object Equals( )` method to return `TRUE` if the specified objects:

- Have the same class type
- Have the same number of elements, and every element contained in one object is contained in the other object

This ensures that collections that are different class instances or contain different class instances are still seen as equal when they represent the same underlying data.

All references to collection objects are specialized in that they return the actual interface they support, allowing you to use the objects without casting.

However, there are no specialized iterators in the collections. All collections support the same `OpenEdge.DataAdmin.Lang.Collections.IIterator` with a `Next( )` method that returns a `Progress.Lang.Object`. So, you must cast the objects returned from a collection with this iterator to the appropriate interface for processing. For more information, see IIterator interface on page 784.

IAreaSet interface

An `IAreaSet` object contains `IArea` elements and defines methods to add and find these elements.

IAreaSet properties

Inherits properties from `IDataAdminCollection`.

IAreaSet methods

Inherits methods from `IDataAdminCollection` and defines the methods listed in the following table.

Table 123: Multi-tenant Maintenance IAreaSet interface methods

Method syntax	Description
`LOGICAL Add ( area AS IArea )`	Adds the area object referenced by *area* to the set and returns TRUE if successful.
`LOGICAL CanFind ( areaname AS CHARACTER )`	Returns TRUE if there is an area object in the set that has its `Name` property set to *areaname*.
`LOGICAL CanFind ( areanum AS INTEGER )`	Returns TRUE if there is an area object in the set that has its `Number` property set to *areanum*.
`LOGICAL Contains ( area AS IArea )`	Returns TRUE if the area object referenced by *area* is in the set.
`IArea Find ( areaname AS CHARACTER )`	Returns the area object from the set that has its `Name` property set to *areaname*. Returns the `Unknown value (?)` if the object is not found.
`IArea Find ( areanum AS INTEGER )`	Returns the area object from the set that has its `Number` property set to *areanum*. Returns the `Unknown value (?)` if the object is not found.

See also:

- IArea interface on page 710
- IDataAdminCollection interface on page 777

IAuthenticationSystemSet interface

An `IAuthenticationSystemSet` object contains `IAuthenticationSystem` elements and defines specialized methods to add, find, and remove authentication systems.

IAuthenticationSystemSet properties

Inherits properties from `IDataAdminCollection`.

IAuthenticationSystemSet methods

Inherits methods from `IDataAdminCollection` and defines the methods listed in the following table.

Table 124: Multi-tenant Maintenance IAuthenticationSystemSet interface methods

Method syntax	Description
`LOGICAL Add ( authn AS IAuthenticationSystem )`	Adds the authentication system object referenced by *authn* to the set and returns TRUE if successful.
`LOGICAL CanFind ( name AS CHARACTER )`	Returns TRUE if there is an authentication system object in the set that has its Name property set to *name*.
`LOGICAL Contains ( authn AS IAuthenticationSystem )`	Returns TRUE if the authentication system object referenced by *authn* is in the set.
`IAuthenticationSystem Find ( name AS CHARACTER )`	Returns the authentication system object from the set that has its Name property set to *name*. Returns the Unknown value (?) if the object is not found.
`LOGICAL Remove( authn AS IAuthenticationSystem )`	Removes the authentication system object referenced by *authn* from the set and returns TRUE if successful.

See also:

- IAuthenticationSystem interface on page 712
- IDataAdminCollection interface on page 777

IDataAdminCollection interface

All the specialized collection interfaces and classes inherit and implement (respectively) the `IDataAdminCollection` interface. The properties and methods of this interface are provided for application use on a sub-interface. Otherwise, the `IDataAdminCollection` interface is not supported for direct access.

The `IDataAdminCollection` interface also defines an `Iterator` property that returns an `IIterator` object, which defines methods for iterating over all, or a subset, of the collection elements.

The properties and methods described in the following sections are supported for use in your application code. `IDataAdminCollection` also defines other properties and methods not described here that are reserved for OpenEdge use only.

IDataAdminCollection properties

Defines the properties listed in the following table.

Table 125: Multi-tenant Maintenance IDataAdminCollection interface properties

Property name	Data type	Access	Description
`ContextId`	`CHARACTER`	Read-only	Returns the context for the instance. Used in the `Equals( )` method to ensure that compared entities with the same keys and underlying data are identified as equal. A blank value means the object has no context (yet), is new, and is not attached, in which case `Equals( )` is only `TRUE` if it compares object references to the same `IDataAdminCollection` instance.
`HasChanges`	`LOGICAL`	Read-only	Returns `TRUE` if one or more attached entity objects have changed since they were attached.
`Service`	`IDataAdminService`	Read-only	Returns the service from which the object was retrieved.
`SourceId`	`CHARACTER`	Read-only	Returns the source for the instance. Returns the same as `ContextId` if this collection is for the entire context.A blank value means the object has no source (yet).

IDataAdminCollection methods

Defines the methods listed in the following table.

Table 126: Multi-tenant Maintenance IDataAdminCollection interface methods

Method syntax	Description
`IIterator Iterator ( filter AS CHARACTER )`	Returns a filtered and/or sorted iterator (`IIterator`) object that provides the selected entities from the collection, where *`filter`* is a query expression on the properties of the entities in the collection. Property expressions and their constituent elements (including operators) **must** be separated by space, and values **must** be quoted. For more information on filtering and sorting entities from a collection, see Filtering and sorting on page 227. For more information on the returned `IIterator` object, see IIterator interface on page 784.
`VOID Export ( )`	Exports the entity collection data to a JSON file named for the sub-interface entity collection, for instance, `areaset.json` or `tenantset.json` for an `IAreaSet` or `ITenantSet` object, respectively.[19]
`VOID Export ( cFilename AS CHARACTER )`	Export entity data in the collection to the file specified by *`cFilename`*.[19]
`VOID ExportLastSaved ( cFilename AS CHARACTER )`	Export entity data in the collection to the file specified by *`cFilename`*.[19] Use only after calling `Import( )` and the related `Create*( )` or `Update*( )` service method.
`VOID ExportLastSavedTree ( cFilename AS CHARACTER )`	Export entity data in the collection with all child collections to the file specified by *`cFilename`*.[19] Use only after calling `Import( )` and the related `Create*( )` or `Update*( )` service method.
`VOID ExportTree ( cFilename AS CHARACTER )`	Export entity data in the collection with all child collections to the file specified by *`cFilename`*.[19]
`VOID ExportTree ( cFilename AS CHARACTER , pcCollections AS CHARACTER )`	Export entity data in the collection with specified child collections to the file specified by *`cFilename`*.[19] Child collections are specified in *`pcCollections`* as a comma-separated list of entity names, for instance, `"Domains,TenantGroups"`, depending on the collection.
`VOID Import ( cFilename AS CHARACTER )`	Import entity data to the collection from the file specified by *`cFilename`*. (The file must exist.)[19]

Method syntax	Description
`VOID ImportTree ( cFilename AS CHARACTER )`	Import entity data to the collection, including child collections, from file specified by *cFilename*. (The top-level entity must exist.)[19]

See also:

- IDataAdminCollection interface on page 777
- IDataAdminElement interface on page 716

IDomainSet interface

An `IDomainSet` object contains `IDomain` elements and defines specialized methods to add, find, and remove domains.

IDomainSet properties

Inherits properties from `IDataAdminCollection`.

IDomainSet methods

Inherits methods from `IDataAdminCollection` and defines the methods listed in the following table.

Table 127: Multi-tenant Maintenance IDomainSet interface methods

Method syntax	Description
`LOGICAL Add ( domain as IDomain )`	Adds the domain object referenced by *domain* to the set and returns TRUE if successful.
`LOGICAL CanFind ( name as CHARACTER )`	Returns TRUE if there is a domain object in the set that has its Name property set to *name*.
`LOGICAL Contains ( domain as IDomain )`	Returns TRUE if the domain object referenced by *domain* is in the set.
`IDomain Find ( name as CHARACTER )`	Returns the domain object from the set that has its Name property set to *name*. Returns the Unknown value (?) if the object is not found.
`LOGICAL Remove ( domain as IDomain )`	Removes the domain object referenced by *domain* from the set and returns TRUE if successful.

[19] Currently supports JSON (.json) files only.

See also:

- IDataAdminCollection interface on page 777
- IDomain interface on page 718

IExtentList interface

An `IExtentList` object contains the extents of an area and defines specialized methods to add (`Add( )`) or return (`Get( )`) an extent. A method to remove extents (`Remove( )`) can only be used on extents that have been added and not committed by the service. This interface is exposed through the `Extents` property on the `IArea` interface.

IExtentList properties

Inherits properties from `IDataAdminCollection`.

IExtentList methods

Inherits methods from `IDataAdminCollection` and defines the methods listed in the following table.

Table 128: Multi-tenant Maintenance IExtentList interface methods

Method syntax	Description
`LOGICAL Add ( pExtent AS IExtent )`	Adds the extent object referenced by *`pExtent`* to the list and returns `TRUE` if successful.
`IExtent Get ( pNum as INTEGER )`	Returns the extent object from the list that has its `Number` property set to *`pNum`*. Returns the Unknown value (?) if the object is not found.
`INTEGER IndexOf ( pExtent AS IExtent )`	Returns the index in this list of the extent object referenced by *`pExtent`*.
`LOGICAL Remove ( pExtent AS IExtent )`	Removes the extent object referenced by *`pExtent`* from the list and returns `TRUE` if successful. Can only execute if the extent is not committed to the database using the `CreateArea*( )` or `UpdateArea*( )` service methods.

See also:

- IArea interface on page 710
- IDataAdminCollection interface on page 777
- IExtent interface on page 721

IFieldPermissionSet interface

A IFieldPermissionSet object contains IFieldPermission elements and defines methods to find these elements in the set. The interface is exposed through the FieldPermissions property on the ITablePermission interface.

IFieldPermissionSet properties

Inherits properties from IDataAdminCollection.

IFieldPermissionSet methods

Inherits methods from IDataAdminCollection and defines the methods listed in the following table.

Table 129: Multi-tenant Maintenance IFieldPermissionSet interface methods

Method syntax	Description
LOGICAL CanFind (*fieldName* as CHARACTER)	Returns TRUE if there is a field permissions object in the set that has its Name property set to *fieldName*.
LOGICAL Contains (*fieldImpl* as IFieldPermission)	Returns TRUE if the field permissions object referenced by *fieldImpl* is in the set.
IFieldPermission Find (*fieldName* as CHARACTER)	Returns the field permissions object from the set that has its Name property set to *fieldName*. Returns the Unknown value (?) if the object is not found.

See also:

- IDataAdminCollection interface on page 777
- IFieldPermission interface on page 723
- ITablePermission interface on page 754

IFieldSet interface

An IFieldSet object contains field elements and defines methods to find fields in the set. The interface is exposed through the LOBFields property on the ITable interface to return LOB field elements.

IFieldSet properties

Inherits properties from `IDataAdminCollection`.

IFieldSet methods

Inherits methods from `IDataAdminCollection` and defines the methods listed in the following table.

Table 130: Multi-tenant Maintenance IFieldSet interface methods

Method syntax	Description
`LOGICAL CanFind ( fieldName as CHARACTER )`	Returns TRUE if there is a field object in the set that has its Name property set to *fieldName*.
`LOGICAL Contains ( fieldImpl as IField )`	Returns TRUE if the field object referenced by *fieldImpl* is in the set.
`IField Find ( fieldName as CHARACTER )`	Returns the field object from the set that has its Name property set to *fieldName*. Returns the Unknown value (?) if the object is not found.

See also:

- IDataAdminCollection interface on page 777
- IField interface on page 722
- ITable interface on page 751

IIndexSet interface

The `IIndexSet` interface contains index elements for an index and defines methods to find indexes in the collection. The interface is exposed through the `Indexes` property on the `ITable` interface.

IIndexSet properties

Inherits properties from `IDataAdminCollection`.

IIndexSet methods

Inherits methods from `IDataAdminCollection` and defines the methods listed in the following table.

Table 131: Multi-tenant Maintenance IIndexSet interface methods

Method syntax	Description
`LOGICAL CanFind ( indexName as CHARACTER )`	Returns TRUE if there is an index object in the set that has its `Name` property set to *`indexName`*.
`LOGICAL Contains ( indexImpl as IIndex )`	Returns TRUE if the index object referenced by *`indexImpl`* is in the set.
`IIndex Find ( indexName as CHARACTER )`	Returns the index object from the set that has its `Name` property set to *`indexName`*. Returns the `Unknown value (?)` if the object is not found.

See also:

- IDataAdminCollection interface on page 777
- IIndex interface on page 725
- ITable interface on page 751

IIterator interface

The `IIterator` interface is in the `OpenEdge.DataAdmin.Lang.Collections` package, along with all the other collection interfaces. This interface provides access to the methods of a general iterator object that iterates over the elements in an entity collection. You can return an instance of `IIterator` from any entity collection using the `Iterator( )` method on the `IDataAdminCollection` interface (and any of its sub-interfaces). Depending on a filter argument that you can pass to an overload of the method, this `IIterator` object can return all or a specified subset of the elements in the collection, and in sorted order, if specified. The iterator returns collection elements as `Progress.Lang.Object` instances. You must then cast each instance to the appropriate interface type for entity access.

IIterator methods

Defines the methods listed in the following table.

Table 132: Multi-tenant Maintenance IIterator interface methods

Method syntax	Description
`LOGICAL HasNext ( )`	Returns TRUE if the iterator has another collection element to return.
`Progress.Lang.Object Next ( )`	Returns the next collection element as an `Object`.
`LOGICAL Remove ( )`	Removes the most recently returned element (by an optional `Next( )` call) from the collection and returns TRUE if successful.

Examples for using the Iterator method

The following code illustrates the use of a collection and iterator, in this case a tenant collection and the iterator used to loop through and allocated tenants:

```
define variable tenantSet as ITenantSet no-undo.
define variable tenant as ITenant no-undo.
define variable iterator as  IIterator no-undo.

tenantSet = service:GetTenants().
iterator = tenantSet:Iterator().
do while iterator:HasNext():
  tenant = cast(iterator:Next(), ITenant).
  If tenant:IsAllocated = false and tenant:Type = "Regular" then
  tenant:Allocate().
end.
service:UpdateTenants(TenantSet).
```

The `Iterator( )` method on `IDataAdminCollection` is overloaded with a parameter to allow the returned iterator to be filtered, sorted, or both. The following code illustrates the use of the iterator filter to achieve the same result as in the previous example:

```
define variable tenantSet as ITenantSet no-undo.
define variable tenant as ITenant no-undo.
define variable iterator as  IIterator no-undo.

tenantSet = service:GetTenants().
iterator = tenantSet:Iterator("IsAllocated = 'false' and type = 'Regular'").
do while iterator:HasNext():
  tenant = cast(iterator:Next(), ITenant).
  tenant:Allocate().
end.
service:UpdateTenants(TenantSet).
```

For more information on filtering and sorting entity collections with the `IDataAdminCollection` iterator, see Filtering and sorting on page 227.

See also:

- IDataAdminCollection interface on page 777

IPartitionCollection interface

An `IPartitionCollection` object contains partitions and is accessed through the `Partitions` property on the `ISchema` interface. This property is only accessible on an `ISchema` with data definitions after the definitions have been loaded in the database using the service `LoadSchemaChanges( )` or `UpdateSchemaChanges( )` method. For more information on these methods, see Schema (reading, loading, and updating) on page 219.

IPartitionCollection properties

Inherits properties from `IDataAdminCollection`.

IPartitionCollection methods

Inherits methods from `IDataAdminCollection` and defines the methods listed in the following table.

Table 133: Multi-tenant Maintenance IPartitionCollection interface methods

Method syntax	Description
`LOGICAL CanFindGroup ( pcGroup AS CHARACTER , pcTable AS CHARACTER , pcField AS CHARACTER , pcIndex AS CHARACTER , pcCollation AS CHARACTER[20] )`	Returns TRUE if there is a partition object in the collection for the tenant group and an appropriate combination of table, field, and index, each specified by name in the respective parameters to identify a single partition, using a blank value as needed.[21]
`LOGICAL CanFindTenant ( pcTenant AS CHARACTER , pcTable AS CHARACTER , pcField AS CHARACTER , pcIndex AS CHARACTER , pcCollation AS CHARACTER[20] )`	Returns TRUE if there is a partition object in the collection for the tenant and an appropriate combination of table, field, and index, each specified by name in the respective parameters to identify a single partition, using a blank value as needed.[21]
`LOGICAL Contains ( partition AS IPartition )`	Returns TRUE if the partition object referenced by *partition* is in the collection.
`IPartition FindGroup ( pcGroup AS CHARACTER , pcTable AS CHARACTER , pcField AS CHARACTER , pcIndex AS CHARACTER , pcCollation AS CHARACTER[20] )`	Returns the partition object from the collection for the tenant group and an appropriate combination of table, field, and index, each specified by name in the respective parameters, using a blank value as needed.[21] Returns the Unknown value (?) if the partition object is not found.
`IPartition FindTenant ( pcTenant AS CHARACTER , pcTable AS CHARACTER , pcField AS CHARACTER , pcIndex AS CHARACTER , pcCollation AS CHARACTER[20] )`	Returns the partition object from the collection for the tenant and an appropriate combination of table, field, and index, each specified by name in the respective parameters to identify a single partition, using a blank value as needed.[21] Returns the Unknown value (?) if the partition object is not found.

See also:

- IDataAdminCollection interface on page 777

[21] For instance, to identify the partition for a table and a tenant, set *pcTenant* to the name of the tenant (such as "EastSports"), then set *pcTable* to the name of the table (such as "Customer") and set *pcField* and *pcIndex* to the empty string (""); to identify the partition for a field in the table, set *pcField* to the name of a field in the specified table (such as "CustNum") and leave *pcIndex* set to the empty string.

[20] Reserved for future use. In the current release, set to the empty string ("").

- IPartition interface on page 727
- IPartitionMap interface on page 787
- IPartitionOwner interface on page 729
- IPartitionPolicyDetailSet interface on page 789

IPartitionMap interface

An `IPartitionMap` object contains partitions for a tenant or tenant group or partition policy detail mapped to an `ISchemaElement` key of type `ITable`, `IIndex`, or `IField` and is accessed though the `Partitions` property on an `ITenant` object or `ITenantGroup` object or a `IPartitionPolicyDetail` object.

IPartitionMap properties

Inherits properties from `IDataAdminCollection`.

IPartitionMap methods

Inherits methods from `IDataAdminCollection` and defines the methods listed in the following table.

Table 134: Multi-tenant Maintenance IPartitionMap interface methods

Method syntax	Description
`LOGICAL Allocate( )`	Allocates all the partitions in the map.
`LOGICAL ContainsKey ( obj AS ISchemaElement )`	Returns TRUE if a partition object is mapped to a table, index, or field key specified by *obj*.
`LOGICAL ContainsValue ( partition AS IPartition )`	Returns TRUE if the partition object referenced by *partition* is in the map.
`IPartition FindField ( pcTable AS CHARACTER , pcField AS CHARACTER )`	Returns the partition object from the map for the `IField` object whose `Name` property is set to *pcField* and represents a field in the table object whose `Name` property is set to *pcTable*. Returns the Unknown value (?) if the object is not found.
`IPartition FindIndex ( pcTable AS CHARACTER , pcIndex AS CHARACTER , pcCollation AS CHARACTER )`	Returns the partition object from the map for the `IIndex` object whose `Name` property is set to *pcIndex* and represents an index in the table object whose `Name` property is set to *pcTable* and with the collation specified by *pcCollation*. Returns the Unknown value (?) if the object is not found.
`IPartition FindTable( pcTable AS CHARACTER )`	Returns the partition object from the map for the `ITable` object whose `Name` property is set to *pcTable*. Returns the Unknown value (?) if the object is not found.
`IPartition Get( obj AS ISchemaElement )`	Returns the partition object that is mapped to a table, index, or field key specified by *obj*. Returns the Unknown value (?) if the object is not found.

See also:

- IDataAdminCollection interface on page 777
- IField interface on page 722
- IIndex interface on page 725
- IPartition interface on page 727
- IPartitionCollection interface on page 785
- ISchemaElement interface on page 746
- IPartitionPolicyDetailSet interface on page 789
- ITable interface on page 751
- ITenant interface on page 755

- ITenantGroup interface on page 761

IPartitionPolicyDetailSet interface

An `IPartitionPolicyDetailSet` object contains `IPartitionPolicyDetail` elements and inherits the standard collection interface of the API (`IDataAdminCollection`). The `IPartitionPolicyDetailSet` objectdefines specialized `CanFind` and `Find` methods to locate a detail policy instance in the collection using the name key.

Note: The `Find` methods that accept a character extent of values expect the values to be formatted according to the session formats for dates and numerics.

IPartitionPolicyDetailSet properties

Inherits properties from `IDataAdminCollection`.

IPartitionPolicyDetailSet methods

Inherits methods from `IDataAdminCollection` and defines the methods listed in the following table.

Table 135: Table Partition Maintenance IPartitionPolicySet interface methods

Method syntax	Description
`LOGICAL CanFind ( name AS CHARACTER )`	Returns `TRUE` if the collection contains a partition policy detail with the specified *name*.
`LOGICAL CanFind ( name AS CHARACTER )`	Returns `TRUE` if the collection contains a partition policy detail with the specified *name*.
`IPartitionPolicyDetail Find ( name AS CHARACTER )`	Returns the partition policy detail object from the set that has its policy detail `policy detail name` property set to *name*. Returns the Unknown value (?) if the object is not found.
`IPartitionPolicyDetail Find ( partition-id AS INTEGER )`	Returns the partition policy object from the set that has its `integer key` property set to *partition-id*. Returns the Unknown value (?) if the object is not found.
`IPartitionPolicyDetail FindLastByValue ( )`	Returns the character partition policy detail by value. The collection defaults to be sorted by the name key.
`IPartitionPolicyDetail FindLastByValue ( value AS * )`	Returns the single field partition policy detail that corresponds to the value, and of the same data type. (* = all data types) This allows the user to find the policy detail that is used to store data for a specific value when the policy is defined as a range.
`IPartitionPolicyDetail FindLastByValue ( values AS CHARACTER [] )`	Returns the character values of a multi-field partition policy detail that corresponds to the passed values. Returns the Unknown value (?) (or throws an error) if the number of extents do not match with the number of fields in the partition policy.

See also:

- IDataAdminCollection interface on page 777
- IUserPermission interface on page 769

IPartitionPolicySet interface

An `IPartitionPolicySet` object contains `IPartitionPolicy` elements and inherits the standard collection interface of the API (`IDataAdminCollection`). The `IPartitionPolicySet object` defines specialized `CanFind` and `Find` methods to locate a policy instance in the collection using the name key.

IPartionPolicySet properties

Inherits properties from `IDataAdminCollection`.

IPartionPolicySet methods

Inherits methods from `IDataAdminCollection` and defines the methods listed in the following table.

Table 136: Table Partition Maintenance IPartitionPolicySet interface methods

Method syntax	Description
`LOGICAL CanFind ( name AS CHARACTER )`	Returns `TRUE` if the collection contains a partition policy with the specified *name*.
`IPartitionPolicy Find ( name AS CHARACTER )`	Returns the partition policy object from the set that has its `name` property set to *name*. Returns the Unknown value (?) if the object is not found.

See also:

- IDataAdminCollection interface on page 777
- IPartitionPolicyDetailSet interface on page 789

ISchemaPartitionMap interface

An `ISchemaPartitionMap` object contains partitions for a schema element mapped to an `ITenant` or `ITenantGroup` key and is accessed through the `Partitions` property on the associated `ITable`, `IIndex`, or `IField` object.

ISchemaPartitionMap properties

Inherits properties from `IDataAdminCollection`.

ISchemaPartitionMap methods

Inherits methods from `IDataAdminCollection` and defines the methods listed in the following table.

Table 137: Multi-tenant Maintenance ISchemaPartitionMap interface methods

Method syntax	Description
`LOGICAL ContainsKey ( pTenant AS ITenant )`	Returns TRUE if a partition object is mapped to the tenant specified by *pTenant*.
`LOGICAL ContainsValue ( partition AS IPartition )`	Returns TRUE if the partition object referenced by *partition* is in the map.
`LOGICAL ContainsKey ( pGroup AS ITenantGroup )`	Returns TRUE if a partition object is mapped to the tenant group specified by *pGroup*.
`IPartition Get ( pElm AS IDataAdminElement)`	Returns the partition object that is mapped to the `IDataAdminElement` specified by *pElm*, which must be an `ITenant` or `ITenantGroup`. Returns the Unknown value (?) if the object is not found.
`IPartition Get ( pGroup AS ITenantGroup )`	Returns the partition object that is mapped to the `ITenantGroup` specified by *pGroup*. Returns the Unknown value (?) if the object is not found.
`IPartition Get ( pTenant AS ITenant )`	Returns the partition object that is mapped to the `ITenant` specified by *pTenant*. Returns the Unknown value (?) if the object is not found.
`IPartition FindGroup ( pcName AS CHARACTER )`	Returns the partition object from the map for the tenant group object that has its `Name` property set to *pcName*. Returns the Unknown value (?) if the object is not found.
`IPartition FindTenant ( pcName AS CHARACTER )`	Returns the partition object from the map for the tenant object that has its `Name` property set to *pcName*. Returns the Unknown value (?) if the object is not found.
`LOGICAL AllocateTenants ( )`	Allocates all delayed partitions for all tenants.Does not allocate partitions belonging to a tenant group.Returns TRUE if any partition was allocated.
`LOGICAL AllocateTenants ( pcAllocationState AS CHARACTER )`	Allocates tenant partitions for the parent schema element (`ITable`, `IIndex`, or `IField`). Does not allocate partitions belonging to a tenant group. The *pcAllocationState* parameter identifies the state of partitions to allocate: `"All"`, `"Delayed"`, or `"None"`. Raises an error if the specified allocation state is not valid.Returns TRUE if any partition was allocated.

Method syntax	Description
`LOGICAL AllocateGroups ( )`	Allocates all delayed partitions for all tenant groups. Does not allocate partitions belonging to a tenant. Returns `TRUE` if any partition was allocated.
`LOGICAL AllocateGroups (` `  pcAllocationState AS CHARACTER )`	Allocates tenant group partitions for the parent schema element (`ITable`, `IIndex`, or `IField`). Does not allocate partitions belonging to a tenant. The *`pcAllocationState`* parameter identifies the state of partitions to allocate: `"All"`, `"Delayed"`, or `"None"`. Raises an error if the specified allocation state is not valid.Returns `TRUE` if any partition was allocated.

See also:

- IDataAdminCollection interface on page 777
- IField interface on page 722
- IIndex interface on page 725
- IPartition interface on page 727
- IPartitionCollection interface on page 785
- IPartitionMap interface on page 787
- ITable interface on page 751
- ITenant interface on page 755
- ITenantGroup interface on page 761

ISequenceSet interface

An `ISequenceSet` object contains sequence elements that can be multi-tenant enabled and defines methods to find add and remove sequences in the set.

ISequenceSet properties

Inherits properties from `IDataAdminCollection`.

ISequenceSet methods

Inherits methods from `IDataAdminCollection` and defines the methods listed in the following table.

Table 138: Multi-tenant Maintenance ISequenceSet interface methods

Method syntax	Description
`LOGICAL Add ( seq AS ISequence )`	Adds the sequence object referenced by *seq* to the set and returns TRUE if successful.
`LOGICAL CanFind ( seqName AS CHARACTER )`	Returns TRUE if there is a sequence object in the set that has its Name property set to *seqName*.
`LOGICAL Contains ( seq AS IDomain )`	Returns TRUE if the sequence object referenced by *seq* is in the set.
`ISequence Find ( seqName AS CHARACTER )`	Returns the sequence object from the set that has its Name property set to *seqName*. Returns the Unknown value (?) if the object is not found.

See also:

- IDataAdminCollection interface on page 777
- ISequence interface on page 748

ISequenceValueMap interface

An `ISequenceValueMap` object contains sequence values for multi-tenant enabled sequences. It is accessed as the `SequenceValues` property on `ISequence` and `ITenant`. The `ISequenceValue` instances of `ITenant` objects are mapped to `ISequence` instances, and `ISequenceValue` instances of `ISequence` objects are mapped to `ITenant` instances, in a many-to-many mapping.

ISequenceValueMap properties

Inherits properties from `IDataAdminCollection`.

ISequenceValueMap methods

Inherits methods from `IDataAdminCollection` and defines the methods listed in the following table.

Table 139: Multi-tenant Maintenance ISequenceValueMap interface methods

Method syntax	Description
`LOGICAL ContainsKey ( pElm AS IDataAdminElement )`	Returns TRUE if the map has the key for an: • `ITenant` when the owner of this collection is an `ISequence` • `ISequence` when the owner of this collection is an `ITenant`
`LOGICAL ContainsValue ( seqVal AS ISequenceValue )`	Returns TRUE if the map contains the sequence value passed by *`seqVal`*.
`ISequenceValue Get ( pElm AS IDataAdminElement)`	Returns the sequence value mapped to the passed `IDataAdminElement` key for an: • `ITenant` when the owner of this collection is an `ISequence` • `ISequence` when the owner of this collection is an `ITenant`

See also:

- IDataAdminCollection interface on page 777
- IDataAdminElement interface on page 716
- ISequence interface on page 748
- ISequenceValue interface on page 749
- ITenant interface on page 755

ITablePermissionSet interface

An `ITablePermissionSet` object contains `ITablePermission` elements and defines methods to find the elements in the collection.

ITablePermissionSet properties

Inherits properties from `IDataAdminCollection`.

ITablePermissionSet methods

Inherits methods from `IDataAdminCollection` and defines the methods listed in the following table.

Table 140: Multi-tenant Maintenance ITablePermissionSet interface methods

Method syntax	Description
`LOGICAL CanFind ( tblName AS CHARACTER )`	Returns TRUE if there is a table permissions object in the set that has its Name property set to *tblName*.
`LOGICAL Contains ( tblImpl AS ITablePermission )`	Returns TRUE if the table permissions object referenced by *tblImpl* is in the set.
`ITablePermission Find ( tblName AS CHARACTER )`	Returns the table permissions object from the set that has its Name property set to *tblName*. Returns the Unknown value (?) if the object is not found.

See also:

- IDataAdminCollection interface on page 777
- ITablePermission interface on page 754

ITableSet interface

An `ITableSet` object contains table elements that can be multi-tenant enabled and define methods to find tables in the collection.

ITableSet properties

Inherits properties from `IDataAdminCollection`.

ITableSet methods

Inherits methods from `IDataAdminCollection` and defines the methods listed in the following table.

Table 141: Multi-tenant Maintenance ITableSet interface methods

Method syntax	Description
`LOGICAL CanFind ( tblName AS CHARACTER )`	Returns TRUE if there is a table object in the set that has its `Name` property set to *`tblName`*.
`LOGICAL Contains ( tblImpl AS ITable )`	Returns TRUE if the table object referenced by *`tblImpl`* is in the set.
`ITable Find ( tblName AS CHARACTER )`	Returns the table object from the set that has its `Name` property set to *`tblName`*. Returns the Unknown value (?) if the object is not found.
`VOID ImportTree ( pcFilename AS CHARACTER , pcCollections AS CHARACTER )`	Import table data to the collection, including child collections specified by *`pcCollections`*, from file specified by *`pcFilename`*. (The top-level table must exist.)[22] Child collections are specified in *`pcCollections`* as a comma-separated list of entity names, for instance, `"Fields,Indexes"`.

See also:

- IDataAdminCollection interface on page 777
- ITable interface on page 751

ITenantGroupSet interface

An `ITenantGroupSet` object contains `ITenantGroup` elements and defines methods to add, find, and remove tenant groups.

This collection is accessible from the DataAdminService. This collection is also accessible through the `TenantGroups` property on `ITenant`. In this case, it represents all the tenant groups in which the tenant, specified by the `ITenant` object, is a member. Adding and removing `ITenantGroup` objects to and from the set adds and removes groups in which the specified tenant is a member.

ITenantGroupSet properties

Inherits properties from `IDataAdminCollection`.

ITenantGroupSet methods

Inherits methods from `IDataAdminCollection` and defines the methods listed in the following table.

[22] Currently supports JSON (.json) files only.

Table 142: Multi-tenant Maintenance ITenantGroupSet interface methods

Method syntax	Description
`LOGICAL Add ( group AS ITenantGroup )`	Adds the tenant group object referenced by *group* to the set and returns TRUE if successful.
`LOGICAL CanFind ( groupName AS CHARACTER )`	Returns TRUE if there is a tenant group object in the set that has its Name property set to *groupName*.
`LOGICAL Contains ( group AS ITenantGroup )`	Returns TRUE if the tenant group object referenced by *group* is in the set.
`ITenantGroup Find ( groupName AS CHARACTER )`	Returns the tenant group object from the set that has its Name property set to *groupName*. Returns the Unknown value (?) if the object is not found.
`LOGICAL Remove ( group AS ITenantGroup )`	Removes the tenant group object referenced by *group* from the set and returns TRUE if successful.

See also:

- IDataAdminCollection interface on page 777
- ITenantGroup interface on page 761

ITenantSet interface

An `ITenantSet` object contains `ITenant` elements and defines methods to add, find, and remove tenants.

This collection is also accessible through the `Tenants` property on `ITenantGroup`. In this case, it represents all the tenants that are members of the tenant group, specified by the `ITenantGroup` object. Adding and removing `ITenant` objects to and from the set adds and removes tenants as members of the group.

ITenantSet properties

Inherits properties from `IDataAdminCollection`.

ITenantSet methods

Inherits methods from `IDataAdminCollection` and defines the methods listed in the following table.

Table 143: Multi-tenant Maintenance ITenantSet interface methods

Method syntax	Description
`LOGICAL Add ( tenant AS ITenant )`	Adds the tenant object referenced by *tenant* to the set and returns TRUE if successful.
`LOGICAL CanFind ( tenantId AS INTEGER )`	Returns TRUE if there is a tenant object in the set that has its Id property set to *tenantId*.
`LOGICAL CanFind ( tenantName AS CHARACTER )`	Returns TRUE if there is a tenant object in the set that has its Name property set to *tenantName*.
`LOGICAL CanFindByExternalId ( extId AS CHARACTER )`	Returns TRUE if there is a tenant object in the set that has its ExternalId property set to *extId*.
`LOGICAL Contains ( tenant AS ITenant )`	Returns TRUE if the tenant object referenced by *tenant* is in the set.
`ITenant Find ( tenantId AS INTEGER )`	Returns the tenant object from the set that has its Id property set to *tenantId*. Returns the Unknown value (?) if the object is not found.
`ITenant Find ( tenantName AS CHARACTER )`	Returns the tenant object from the set that has its Name property set to *tenantName*. Returns the Unknown value (?) if the object is not found.
`ITenant FindByExternalId ( extId AS CHARACTER )`	Returns the tenant object from the set that has its ExternalId property set to *extId*. Returns the Unknown value (?) if the object is not found.
`LOGICAL Remove ( tenant AS ITenant )`	Removes the tenant object referenced by *tenant* from the set and returns TRUE if successful.

See also:

- IDataAdminCollection interface on page 777
- ITenant interface on page 755

IUserFieldPermissionSet interface

An IUserPermissionSet object contains IUserPermssion elements and defines methods to find user permissions. You can retrieve an IUserPermissionSet object through the UserFieldPermissions property on the IUserTablePermission interface.

IUserFieldPermissionSet properties

Inherits properties from `IDataAdminCollection`.

IUserFieldPermissionSet methods

Inherits methods from `IDataAdminCollection` and defines the methods listed in the following table.

Table 144: Multi-tenant Maintenance IUserFieldPermissionSet interface methods

Method syntax	Description
`LOGICAL CanFind ( fieldName AS CHARACTER )`	Returns TRUE if there is a field permissions object in the set that has its Name property set to *fieldName*.
`LOGICAL Contains ( fieldImpl AS IUserFieldPermission )`	Returns TRUE if the field permissions object referenced by *fieldmpl* is in the set.
`IUserFieldPermission Find ( fieldName AS CHARACTER )`	Returns the field permissions object from the set that has its Name property set to *fieldName*. Returns the Unknown value (?) if the object is not found.

See also:

- IDataAdminCollection interface on page 777
- IUserPermission interface on page 769
- IUserTablePermission interface on page 770

IUserPermissionSet interface

An `IUserPermissionSet` object contains `IUserPermssion` elements and defines methods to find user permissions.

IUserPermissionSet properties

Inherits properties from `IDataAdminCollection`.

IUserPermissionSet methods

Inherits methods from `IDataAdminCollection` and defines the methods listed in the following table.

Table 145: Multi-tenant Maintenance IUserPermissionSet interface methods

Method syntax	Description
`LOGICAL CanFind ( userId AS CHARACTER )`	Returns TRUE if there is a user permissions object in the set that has its `Id` property set to *`userId`*.
`LOGICAL Contains ( user AS IUserPermission )`	Returns TRUE if the user permissions object referenced by *`user`* is in the set.
`IUserPermission Find ( userId AS CHARACTER )`	Returns the user permissions object from the set that has its `Id` property set to *`userId`*. Returns the Unknown value (?) if the object is not found.

See also:

- IDataAdminCollection interface on page 777
- IUserPermission interface on page 769

IUserSet interface

An `IUserSet` object contains `IUser` elements and defines methods to add, find, and remove users.

IUserSet properties

Inherits properties from `IDataAdminCollection`.

IUserSet methods

Inherits methods from `IDataAdminCollection` and defines the methods listed in the following table.

Table 146: Multi-tenant Maintenance IUserSet interface methods

Method syntax	Description
`LOGICAL Add ( user AS IUser )`	Adds the user object referenced by *user* to the set and returns TRUE if successful.
`LOGICAL CanFind ( userName AS CHARACTER )`	Returns TRUE if there is a user object in the set identified by *userName*. The value of *userName* depends on the source for the `IUserSet`. If the source is the `Users` property on an `IDomain` object, *userName* must match the `Name` property of an `IUser` object in the set. If the source is the `GetUsers( )` service method, *userName* must match the `Id` property of an `IUser` object in the set.
`LOGICAL Contains ( user AS IUser )`	Returns TRUE if the user object referenced by *user* is in the set.
`IUser Find ( userName AS CHARACTER )`	Returns the user object from the set identified by *userName*. The value of *userName* depends on the source for the `IUserSet`. If the source is the `Users` property on an `IDomain` object, *userName* must match the `Name` property of an `IUser` object in the set. If the source is the `GetUsers( )` service method, *userName* must match the `Id` property of an `IUser` object in the set. The method returns the Unknown value (?) if a user object is not found.

See also:

- IDataAdminCollection interface on page 777
- IUser interface on page 766

IUserTablePermissionSet interface

An `IUserTablePermissionSet` object contains `IUserTablePermssion` elements and defines methods to find user permissions. You can retrieve an `IUserTablePermissionSet` object from the DataAdminService using the `GetTablePermissions( )` method, and also through the `UserTablePermissions` property on the `IUserPermission` interface.

IUserTablePermissionSet properties

Inherits properties from `IDataAdminCollection`.

IUserTablePermissionSet methods

Inherits methods from `IDataAdminCollection` and defines the methods listed in the following table.

Table 147: Multi-tenant Maintenance IUserTablePermissionSet interface methods

Method syntax	Description
`LOGICAL CanFind ( userId AS CHARACTER )`	Returns TRUE if there is a user permissions object in the set that has its `Id` property set to *`userId`*.
`LOGICAL Contains (` `  user AS IUserPermission )`	Returns TRUE if the user permissions object referenced by *`user`* is in the set.
`VOID ExportNormalized (` `  cFilename AS CHARACTER )`	Export entity data in the collection to the file specified by *`cFilename`*.[23] This method exports data in a more normal JSON format than the proprietary ProDataSet JSON format used by other Export*( ) and Import*( ) methods.
`IUserPermission Find ( userId AS CHARACTER` `  )`	Returns the user permissions object from the set that has its `Id` property set to *`userId`*. Returns the Unknown value (?) if the object is not found.

See also:

- IDataAdminCollection interface on page 777
- IUserPermission interface on page 769

[23] Currently supports JSON (.json) files only.

Index

^ (caret), I/O statements *315*
~ (tilde), input files *321*

A

ABL API for CDC *645*
ABL API for Multi-tenant Maintenance
    collection interface reference *773*
    entity interface reference *707*
ABL callbacks *166*, *171–172*
ABL data types
    converting to C data types *394*
    converting to COM data types *611*
ABL data, accessing with HLC library functions *391*
ABL key functions, ActiveX controls *563*
Abnormal termination *398*
Accessing help *456*
Accessing OCXs at run time *559*
ActiveX
    Automation *543*
    Automation objects *329*
    collections, navigating *536*
    controls *329*
ActiveX automation, See Automation
ActiveX controls (OCXs)
    accessing
        at run time *559*
    assigning colors *329*
    assigning fonts *329*
    control container *552–553*
    creating instances *556*
    CSComboBox control *577*
    CSSpin control *578*
    defined *428*, *551*
    defining instances
        design-time properties *557*
        invisible controls *559*
        location *558*
        naming *557*
        size *558*
        tab order *559*
    design mode, defined *431*
    DLL files *431*
    encapsulation *553*
    event procedures in the AppBuilder *575*
    events
        defined *430*
    examples
        i-ocx1.w *552*
    files
        design time *431*
ActiveX controls (OCXs) *(continued)*
    files *(continued)*
        run time *431*
    handle connections *562*
    installed with OpenEdge *577*
    instance files (.wrx) *431*
    instantiating at run time *559*
    license information *431*
    loading with ABL *559*
    locating in viewer *540*
    managing at run time
        ABL key functions *563*
        releasing resources *563*
        tab order and Z order *563*
    programming in the AppBuilder *570*
    properties, design-time vs. run-time *556*
    PSTimer control *578*
    requirements
        design mode *431*
        run mode *431*
    restrictions *556*
    run mode, defined *431*
    sources *430*
Add additional index entries here (if needed)
    Nest index terms for sub entries *666*
ADD-EVENTS-PROCEDURE method *569*
Adding records *49*
Advise links
    creating *516*
    defined *515*
    multiple conversations *517*
    removing
        example *518*
AfterSetIdentity *172*
Aggregate functions
    TOTAL BY *352*
Aliases
    compiling procedures *46*
    creating in applications *45*
    DICTDB *135*
    logical database name *44*
    shared record buffers *47*
ALTERNATE-KEY option
    SYSTEM-HELP statement *457*
ANYWHERE option *286*, *290*
API entities *646*
API for Multi-tenant Maintenance, ABL *205*
API service methods *646*
APIs, Audit Policy Management
    access *256*
AppBuilder *452*, *455*

AppBuilder and ActiveX controls
control loading and instantiation 572
control-frame instantiation 570
default handles 570
event procedures 575
initialization 575
resource definitions 570
standard procedures 577
applhelp.p 463
Application environment management 340
Application events
auditing 246
Application security
activities-based authorization 159
authorizing user IDs at run time 155
CAN-DO function 160
creating, permissions table 159
cryptography 175
secure database connections 36
types 97
using the CAN-DO function 156
Application trusted domain registry
ABL for building 124
Application user ID
defined 105
managing 145
Archiving
audit 255
arrays 613
AS Datatype option, COM object syntax 530
AS option
DEFINE PARAMETER statement 487
ASC function, code page conversion 317
ASSIGN statement 47
Attributes
AVAILABLE-FORMATS 438
control-frame widget 554
DDE-ERROR 511
DDE-ID 512
DDE-ITEM 512
DDE-NAME 512
DDE-TOPIC 512
ITEMS-PER-ROW 438
MULTIPLE 439
NUM-FORMATS 439
TYPE 439
VALUE 439
Audit data tables 257
Audit Policy Management
API access 256
APIs
application server usage 619
caching 630
exporting to XML 636
generic utility 620
importing from XML 637
Audit Policy Management *(continued)*
APIs *(continued)*
overview 618
procedure file reference 618
stand-alone procedures 641
Audit policy tables 256
AUDIT-CONTROL system handle 247
AUDIT-ENABLED function 245
Auditing
See also Auditing ABL
defined 97
developing tools
audit archiving 257
audit configuration 255
recording application events
ABL for 247
planning 246
See also Auditing ABL
Auditing ABL
See also Auditing
asserting auditing ID 243
audit archiving
coding requirements 258
required privileges and security 259
audit configuration
APIs for 256
privileges required 257
audit data table access 257
audit event context
application context 251
application design 253
audit event groups 252
client login session 249
management 248
audit policy table access 256
generating application events 247
recording an application event 248
security
privileges 245
verifying audit enablement 245
See also Auditing
Auditing ID
asserting
application design 244
from database connection ID 243
from OpenEdge session ID 244
configuring 242
defined 105
establishing 241
identity management 154
Auditing privileges 245
AuthenticateUser 172
Authentication
database connection
examples 135
defined 97

Authentication *(continued)*
models *98*
OpenEdge-supported procedure *135*
Authentication systems *98*, *166*
Authorization
compile-time
defined *97*
defined *97*
examples *156*
run-time
activity access *159*
defined *97*
procedure execution *156*
table and field access *165*
schema
defined *97*
Auto-connect, connecting databases *39*
Automation
See also Automation objects
defined *543*
requirements *543*
servers, accessing *544*
See also Automation objects
Automation objects
accessing
by name *544*
by name and file *546*
implied object and file *547*
named top-level object *545*
compared to DDE *427*
defined *543*
locating in viewer *539*
registry flags *544*
releasing *548*
requirements *431*
sources *430*
top-level, defined *543*
Automation, ActiveX Automation objects *329*
AVAILABLE-FORMATS attribute *438*

B

Base fields *363*
Base64 encoding *154*
BASE64-DECODE function
data size decrease *188*
managing crypto data with *187*
BASE64-ENCODE function
data size increase *188*
managing crypto data with *187*
Batch processing *315*
BEGIN-EVENT-GROUP method
usage *252*
BGCOLOR, widget assignment *327*
Blocks
editing *290*
BROWSE statement *47*
Buffers
record *47*
screen *47*
Building HLC executables, See HLC executable
Bulk CDC policies *652*
Buttons, mouse *284*
BY-POINTER type option *532*
BY-VARIANT-POINTER type option *532*

C

C functions
declaration with FUNCTEST *388*
HLC functions *391*
naming *389*
passing error codes *396*
portability *390*
returning error codes *389*
writing *388*
C language data types, converting to ABL data types *394*
CALL statement
shared buffer *395*
syntax *377*
using *377*
Calling conventions for DLL routines *487*
Calling help *456*
CAN-DO function
application security *160*
using to authorize user IDs *156*
using with ROLES attribute *156*
CDC Dump Utility *661*
CDCOperation enumeration *667–668*
CDCTrackingHelper class *667–668*
CDECL option, PROCEDURE statement *487*
CHARACTER data type
running DLL routines *499*
when modified by shared library routine *497*
Character interfaces
colors
reserved values *330*
widget states *330*
specifying colors *326*
CHARACTER parameters, passing to shared library routines *497*
Character set management, See Code page conversion
Checking data entry for keys *289*
CHOOSE event *460*, *462*
CHR function
code page conversion *317*
CLEAR-APPL-CONTEXT method
usage *251*
Client login session
audit event context management *249*
recording *133*
starting and maintaining *134*

Client-principal object
attributes
read-only *112*
auditing usage *249*
methods *118*
multi-tier applications *147*
Client-principal objects *99*
Clipboard, See System clipboard
Clipboard examples
i-clpbrd.p *443*
i-clpmul.p *447*
Clipboard output *301*
CLIPBOARD system handle
example procedure
multiple-item transfer *447*
single-item transfer *443*
multiple-item transfers *445*
requirements *424*
single-item transfers *439*
Code examples *651–652*, *654–656*
Code pages
conversion map file *317*
conversions *317*
example of conversion *317*
CODEPAGE-CONVERT function *317*
Coding help calls *456*
See also Help calling interface
Coding help strings *452*
Collection interfaces *773*
Collections, navigating *536*
Color
adding for startup *326*
availability *326*
character interfaces
management *330*
reserved values *330*
widget states *330*
common dialog
example *333*
custom, Windows color dialog *333*
definitions
changing *339*
dynamic *332*
environment files *326*
environments *340*
graphical interfaces
definition access *339*
environment management *340*
management *331*
table access *337*
user changes *332*
inheritance *329*
managing display limitations *342*
numeric, Windows color dialog *333*
OpenEdge defaults *326*
RGB-VALUE function *326*
Color *(continued)*
table access *337*
the registry *326*
visual, Windows color dialog *333*
widget assignment
example *327*
Windows
Palette Manager *331*
Color dialogs
ABL statements *332–333*
Windows example *333*
COLOR-TABLE handle
example *327*, *332*
using *337*
Colors
assigning *327*
assigning to ActiveX controls *329*
modifying *337*
COM data types
alternate names *614*
converting to ABL *610*
COM Object Viewer
accessing Type Libraries *538*
locating
ActiveX controls *540*
Automation objects *539*
methods, properties, and events *540*
COM objects
See also ActiveX controls (OCXs), Automation objects
accessing *429*
ActiveX collections *536*
colors and fonts *534*
compared to widgets
memory management *536*
control-frame *553*
defined *427*
error handling *537*
handles *524*
managing
memory resources *536*
obtaining access *525*
properties and methods *525*
sources *430*
Type Libraries *531*
viewing on-line *538*
See also ActiveX controls (OCXs), Automation objects
COM-HANDLE data type *524*
COM-SELF system handle *568*
COMMAND option
SYSTEM-HELP statement *457*
Common dialogs
ABL statements *332–333*, *335*
color *333*
examples *333*, *335*
font *335*

Compile-time, authorization 97
Component handles
accessing for OCXs 560
ActiveX control connections 562
expressions 527
releasing COM objects 537
validating 534
Configuration
audit 255
CONNECT statement
authentication 135
NO-ERROR option 37, 41
CONNECTED function 41
Connection failures 40
See also Multi-database connection(s)
Connection overhead 42
Connection parameters, See Startup parameters
CONTAINS operator 71
CONTENTS option
SYSTEM-HELP statement 457
CONTEXT option
SYSTEM-HELP statement 457
CONTEXT-POPUP option
SYSTEM-HELP statement 457
Context-sensitive help 456, 462–463
Control container, ActiveX, defined 552
Control sequences 372
Control tables, sequences 82
Control-frame
attributes and properties 554
COM object 553
widget
events 565
Control-Name property, OCX handle access 561
Controllers, Automation, defined 543
Controls
ActiveX controls 329
Conversations, DDE
closing 518
defined 508
end-point 510
exchanging data 515
opening 512
structure 509
using 508
Conversion map files 317
Converting
code pages 317
data
IMPORT statement 321
Cooked terminal I/O mode 396
Copy operations using the clipboard 442
CREATE ALIAS statement 44
CREATE Automation Object statement 544
Create bulk CDC policies 652
Create methods 647–648
CREATE statement 47
Create, read, update, and delete examples 651
Creating CDC table policies 651
Crypto data
database and transport management 187
management requirements 186
size increase 188
storage 187
Cryptography
algorithms
symmetric encryption 177
code examples 183
defined 97
encryption keys
generating 178
setting symmetric 178
generating keys 178
managing crypto data 186
message digests 185
security policy 176
symmetric functions 181
CSComboBox control (OCX) 577
CSSpin control (OCX) 578
CURRENT-VALUE function
syntax 85
CURRENT-VALUE statement
syntax 85
Cursor repositioning
field lists 62
FIND statement
after FIND fetches 58
Custom colors, Windows color dialog 333
Cut operations using clipboard 441

D

Data
cryptography 175
handling statements 47
movement 47
Data Dictionary
creating, permissions table 159
Data entry, monitoring 264, 289
Data integrity 185
Data size 391
Data type conversions
COM object
ABL to COM data types 611
COM to ABL data types 610
features and limitations 610
pointers 612
strategy 610
types not converted 614
HLC 393
Data types
COM alternate names 614

Data types *(continued)*
COM object syntax *530*
COM-HANDLE *524*
DLL RUN parameter lists *499*
parameters, OCX event procedures *567*
RAW *90*
shared library parameter definitions *490*
Database connection ID
managing
from external authentication system *138*
from internal authentication system *135*
Database events
auditing *246*
Database fields
help *450*
help strings *449*
Databases
connect parameters, limits *36*
connection security *36*
multiple connections, See Multi-database connections
Number of Databases (-h) startup parameter *36*
security *97*
DCOLOR
using *330*
widget assignment *327*
widget states *330*
DDE, See Dynamic data exchange
DDE ADVISE statement
using *516*
DDE EXECUTE statement
using *515*
DDE frames
DDE-NOTIFY events *516*
defined *510*, *514*
DDE GET statement
using *516*
DDE INITIATE statement
using *514*
DDE REQUEST statement
using *515*
DDE Return Codes *512*
DDE SEND statement
using *515*
DDE server preparations *512*
DDE TERMINATE statement
using *518*
DDE-ERROR attribute *511*
DDE-ID attribute *512*
DDE-ITEM attribute *512*
DDE-NAME attribute *512*
DDE-NOTIFY events
DDE frames *510*
trapping *516*
DDE-TOPIC attribute *512*
Declaring DLL routines, by number *487*
DECRYPT function
code examples *183*
syntax *181*
Decryption, See Cryptography
Default properties and methods *534*
DEFINE BROWSE statement *454*
DEFINE BUTTON statement *454*
DEFINE FRAME statement *452*
DEFINE IMAGE statement *454*
DEFINE PARAMETER statement for DLLs *487*
DEFINE QUERY statement
SCROLLING option *56*
DEFINE RECTANGLE statement *454*
DEFINE SHARED STREAM statement *311*
See also Streams, shared
DEFINE STREAM statement *309*
Defining shared library parameters *487*
DELETE ALIAS statement *44*
Delete API *650*
Delete methods *650*
DELETE statement *47*, *49*
DELETE WIDGET statement, releasing ActiveX controls *563*
Deleting
field lists *61*
records *49*
Deleting CDC policies *656*
DELIMITER option
EXPORT statement *319*
IMPORT statement *319*, *321*
Demand-driven DDE *515*
Design mode, ActiveX control
requirements *431*
Design-time OCX properties
setting *557*
Dialog boxes
help *449*
DICTDB alias *135*
Directories, HLC *385*
DISCONNECT statement
affect on aliases *44*
Dispatch routine, HLC, See PRODSP dispatch routine
DISPLAY statement *47*
Displaying data *350*
DLL, See Dynamic link library (DLL)
DLL examples
i-dllex1.p *501*
i-dllex2.p *501*
DLL routines
passing NULLs *496*
DLL structures
allocating with DLL routines *497*
freeing memory *415*, *497*
functions for reading *417*
statements for writing *417*
Domain registries *99*

Dump CDC *661*
Dynamic Data Exchange (DDE)
ABL statements *508*
application name *509*
channel number
defined *509*
usage *514*
client, defined *508*
client/server coordination *517*
conversational exchange
defined *510*
types *515*
conversations
closing *518*
defined *508*
exchanging data *515*
hierarchy *509*
multiple *517*
opening *512*
structure *509*
data item name *509*
DDE-NOTIFY events *516*
defined *426*, *507*
demand-driven *515*
event-driven *516*
example procedure *519*
frame attributes *510*
frames
DDE-NOTIFY events *516*
defined *508*, *510*, *514*
initiating *514*
overview *508*
requirements *427*
server
defined *508*
preparation *512*
topic name *509*
using *508*
Dynamic link libraries (DLL)
access to ABL widgets *498*
capabilities *483*
defined *424*, *483*
HWND attribute *498*
NULL parameter values *496*
parameter definition *487*
requirements *425*
using *484*

E

Editing blocks *290*
EDITING blocks *290*
ENABLE statement *47*, *452*
ENCODE function *102*
Encoding UUIDs *154*
ENCRYPT function
code examples *183*
syntax *181*
ENCRYPT-AUDIT-MAC-KEY() method *103*
encrypted passwords *103*
Encryption, See Cryptography
END-ERROR key *286*
END-EVENT-GROUP method
usage *252*
Entity interfaces *707*
Enums *656*
Enums for CDC *656*
Environment files
changing resources *339*
colors and fonts *326*
hints for managing *342*
managing *341*
managing multiple *341*
managing statements *341*
EPI, See External program interfaces (EPIs)
Error codes, returning *389*
Error handling
COM objects *537*
Unix Systems *398*
Event procedures, OCX
coding rules *566*
defined *565*
examples
AppBuilder *575*
parameters
data type conversions *609*
searching at run time *569*
specifying in the UIB *566*
system handles *568*
Event-driven DDE
applications *517*
Events
ActiveX control
defined *430*
locating in viewer *540*
control-frame widget *565*
keyboard *290*
widget and OCX
applying from ABL *568*
Example procedures
Clipboard multiple-transfer *447*
Clipboard single-transfer *443*
DLL for message display *501*
DLL to play sounds *501*
UNIX named pipe *470*
Exchange, conversational
defined *510*
demand-driven
defined *515*
example *515*
using *515*

Exchange, conversational *(continued)*
event-driven
defined *515*
example *517*
using *516*
types *515*
Exiting help from application *464*
EXPORT statement
DELIMITER option *319*
Exporting data *319*
EXTEND mouse button *284*
External authentication systems
managing database connection ID *138*
External program interfaces (EPIs), defined *413*

F

Fetching records
defining record set *49*
order *51*
preselect list *52*
results list *52*
ROWID and RECID *52*
using field lists, See Field lists
FGCOLOR, widget assignment *327*
Field List Disable (-fldisable) startup parameter *62*
Field lists
benefits *59*
cursor repositioning *62*
DataServers *59*, *61–62*
degenerate cases *62*
deletes *61*
deployment *62*
implied entries *61*
multiple queries *62*
server versions *62*
shared queries *60*
specifying *59*
updates *61*
Field policies *660*
Field policy API *647*
Field policy interface properties *660*
Fields
validation *318*
File types *315*
FILE-INFO system handle *315*
Files
ActiveX control *431*
attributes *315*
conversion map *317*
convmap.cp *317*
HLC *385*
information *315*
input *318*
redirecting input *306*
redirecting output *301*
Files *(continued)*
schema cache *93*
FIND statement
repositioning
after FIND fetches *58*
FINDER option
SYSTEM-HELP statement *457*
FOCUS handle, clipboard *439*
Font dialogs
ABL statements *332*, *335*
Windows example *335*
FONT-TABLE handle
example *327*
using *338*
FONT, widget assignment *327*
Fonts
adding for startup *327*
assigning *327*
assigning to ActiveX controls *329*
availability *326*
character interfaces, management *330*
common dialog
example *335*
definitions
changing *339*
environment files *326*
environments *340*
graphical interfaces
definition access *339*
environment management *340*
management *331*
table access *337*
user changes *332*
inheritance *329*
OpenEdge defaults on Windows *327*
table access *337*
widget assignment
example *327*
FOR EACH statement *47*
FORCE-FILE option
SYSTEM-HELP statement *457*
FORM statement *452*
Format phrase *452*
ABL elements, for sockets *580*
Frame backgrounds *364*
Frame headers *364*
FRAME-FIELD function *290*
FUNCTEST macro
C function declaration *388*
syntax *378*
Function keys
changing definitions *282*
monitoring usage *289*
Functions
ASC *317*
CAN-DO *155*, *160*

Functions *(continued)*
CHR *317*
CODEPAGE-CONVERT *317*
ENCODE *102*
ENCRYPT and DECRYPT *181*
FRAME-FIELD *290*
HLC interrupt handling *673*
HLC library *674*
HLC screen display *673*
HLC shared buffer access *672*
HLC shared variable access *671*
HLC timer service routines *673*
KBLABEL *265*
KEYCODE *265*
KEYFUNCTION *265*
KEYLABEL *265*
LASTKEY *286*
OS-GETENV *315*
PAGE-NUMBER *364*
SEARCH *302*
SET-DB-CLIENT *135*
SUBSTRING *701*
USERID *156*

G

Garbage collection, COM objects *536*
Generate policy program *661*
Get CDC policies *654*
Get Filter *649*
Get methods *648–649*
Get policy API *648*
Get Table *649*
GET-BYTE function *90*
GET-datatype functions *416*
GET-KEY-VALUE statement
example *340*
GET-POINTER-VALUE function *420*
GET-RGB-VALUE method, COM objects *535*
GET-SIZE function *415*
GO-ON phrase *286*

H

HANDLE option *491*
handles *312*
Handles
COLOR-TABLE *337*
FILE-INFO *315*
FONT-TABLE *338*
SECURITY-POLICY
application trusted domain registry *124*
cryptography *176*
Help access methods *460*, *463*
Help buttons *462*
Help calling interface *456*
HELP events *463*
Help keys *462–463*
HELP option
SYSTEM-HELP statement *457*
Help strings
coding in the AppBuilder *452*
database fields *450*
dialog boxes *449*
field-level widgets *452*
in code *452*
Help systems
context-sensitive help *456*
Help topics *456*
HELP-TOPIC option
SYSTEM-HELP statement *457*
HEX-DECODE function
managing crypto data with *187*
HEX-ENCODE function
managing crypto data with *187*
Hexidecimal encoding *154*
HLC
common errors *390*
directories *385*
filenames *385*
mapping C functions *378*
Unix Systems *396*
using *376*
HLC datatype conversions *393*
HLC dispatch routine, See PRODSP dispatch routine
HLC executables, building *385*
HLC library functions
calling *394*
example *393*
guidelines for accessing ABL data *391*
interrupt handling *673*
screen display *673*
shared buffer *395*
shared buffer access *672*
shared variable access *671*
timer service routines *673*
HLC library functions described
prockint() *674*
proclear() *674*
procncel() *675*
proevt() *675*
profldix() *676*
promsgd() *676*
prordbc() *677*
prordbd() *678*
prordbi() *679*
prordbl() *680*
prordbn() *681*
prordbr() *682*
prordc() *683*
prordd() *684*
prordi() *685*

HLC library functions described *(continued)*
prordl() *685*
prordn() *686*
prordr() *687*
prosccls() *688*
proscopn() *689*
prosleep() *689*
prowait() *689*
prowtbc() *690*
prowtbd() *691*
prowtbi() *692*
prowtbl() *693*
prowtbn() *693*
prowtbr() *694*
prowtc() *695*
prowtd() *696*
prowti() *697*
prowtl() *698*
prowtn() *699*
prowtr() *699*
Host Language Call interface
requirements *377*
HWND attribute *498*

I

I/O with other processes *314*
Identity
OpenEdge authenticated and default *104*
Identity management
ABL to manage user IDs *106*
application trusted domain registry *124*
application user IDs
managing *145*
auditing IDs
managing *154*
client-principal objects *111*
database connection IDs *135*
multi-tier applications
AppServer operating mode *145*
client-principal object *147*
state-free *151*
stateless *149*
OpenEdge session IDs
security policy *124*
unique object IDs *154*
IMPORT statement
DELIMITER option *319*, *321*
Importing, data *321*
Index cursors *58*
Index-sorted queries *54*
Indexing, word *71*
Initialization files, colors and fonts *326*
Inner joins
defined *63*
example *63*, *66*
Inner joins *(continued)*
mixing with left outer *68*
specifying *64*
Input
changing the source *306*
control character *309*
devices *307*
directory contents *307*
file preparation
IMPORT statement *321*
files *306*
multiple sources *307*, *309*
streams *298*
terminal *307*
working within procedures *297*
INPUT CLOSE statement
QUOTER utility *701*
Input focus *463*
INPUT FROM statement
BINARY option *309*
code page conversion *317*
OS-DIR option *315*
QUOTER utility *701*
redirecting input *307*
INPUT mode option, OCX event parameters *567*
INPUT THROUGH statement
code page conversion *317*
INPUT-OUTPUT mode option
method parameters *532*
INPUT-OUTPUT THROUGH statement
code page conversion *317*
INSERT statement *47*, *49*
Instantiating ActiveX controls *559*
Internal authentication system
managing database connection ID *135*
Internal events
auditing *246*
internationalization, word break tables *71*
Interprocess communications
I/O with other processes *314*
Interrupt handling, HLC library functions *673*
Invisible ActiveX controls *559*
Item property, OCX handle access *562*
ITEMS-PER-ROW attribute
example *447*

J

Joins
See also Inner join, Left outer join
conditions *64*
defined *63*
full outer *68*
inner *63*
left outer *63*
mixing inner and left outer *68*

Joins *(continued)*
nested reads *65*
right outer *68*
specifying *63*
See also Inner join, Left outer join

K

KBLABEL function *265*
Key functions, See ABL key functions
KEY option
SYSTEM-HELP statement *457*
Keyboard events *290*
KEYCODE function *265*
KEYFUNCTION function *265*
KEYLABEL function *265*
Keys
changing the function *282*
codes *266*
END-ERROR *286*
functions *277*, *290*
input *264*
input precedence *264*
labels *266*, *290*
Keystrokes
monitoring *264*, *289*

L

LASTKEY function *286*
Left outer joins
defined *63*
example *64*, *67*
mixing with inner *68*
specifying *64*
Licensed user count, exceeding *36*
LIKE option
connected database *93*
DEFINE PARAMETER statement *487*
LOAD statement
using *341*
LoadControls method, role in OCX instantiation *559*
LOG-AUDIT-EVENT method
usage *248*
_login.p procedure *135*

M

Managing audit event context *248*
Managing COM objects *534*
Memory allocation, C routines *390*
Memory management, COM objects *536*
MEMPTR data type
database fields *491*
referencing strings *497*
MEMPTR data type *(continued)*
running shared library routines *496*
MEMPTR parameters, used to pass character strings *497*
MEMPTR variables
initialization *415*
initializing and uninitializing
DLL routines *497*
obtaining the pointer value *420*
passing string values *497*
reading and writing *416*
using *496*
MENU mouse button *284*
Menus
accessing help *460*
Message digests *185*
See also Cryptography
Method references, COM object
parentheses required *533*
restrictions *533*
Methods and properties, COM object, See Properties and methods, COM object
Methods, COM object
See also Properties and methods, COM object
passing parameters
named *533*
See also Properties and methods, COM object
Mode options
purpose *531*
using *532*
Monitoring keystrokes
EDITING blocks *290*
user interface triggers *290*
Mouse
buttons
EXTEND *284*
MENU *284*
MOVE *284*
events
specifying *285*
input *264*
Mouse buttons, SELECT *284*
MOVE mouse button *284*
Multi-database applications
referencing files and field names *91*
Multi-database connections
ABL functions *41*
CONNECTED function *41*
connection overhead *42*
connection parameters *36*
conserving *42*
considerations at development *39*
disconnecting *43*
failures and disruptions *40*
logical database name *43*
methods list *36*

Multi-database connections *(continued)*
Physical Database Name (-db) startup parameter *37*
positioning references in applications *93*
run-time considerations *40*
Multi-databases
disconnecting *43*
logical database name *43*
Multi-tenancy
ABL support *189*
configuring with ABL *205*
Multiple
input sources *307*
output destinations *302*
MULTIPLE attribute
example *447*
Multiple-item transfers, clipboard
data-based *446*
example *447*
widget-based *446*
MULTIPLE-KEY option
SYSTEM-HELP statement *457*
MULTITASKING-INTERVAL attribute
using *517*

N

NAME attribute, control-frame *557*
Name property, ActiveX controls *557*
Named pipes
accessing *466*, *469*
advantages *468*
applications *466*
capacity *467*
compared to files and unnamed pipes *466*
creating
mknod command *468*
mknod() system call *468*
defined *424*, *465*
deleting
rm command *469*
unlink() system call *469*
disadvantages *468*
examples
i-do-sql.p *470*
i-pipex2.p *470*
I/O synchronization *467*
interleaving messages *467*
message handler procedure *466*
multiple users *467*
operational characteristics *467*
overview *466*
process blocking *467*
request format example *470*
requirements *424*
scenario *466*
Named pipes *(continued)*
UNIX
creating *468*
deleting *469*
examples *470*
using *468*
writing messages *468*
Naming ActiveX controls *557*
Naming control-frames *557*
New methods *646–647*
NEXT-VALUE function
syntax *85*
NO-ERROR option
CONNECT statement *37*
NO-RETURN-VALUE option
method reference *529*
Non-OpenEdge databases *36*
NUM-FORMATS attribute *439*
Number of Databases (-h) startup parameter *36*
Number of Users (-n) startup parameter *40*
Numeric colors, Windows color dialog *333*

O

Objects
Automation *543*
COM *427*
OCX, See ActiveX controls (OCXs)
OCX event triggers, See Event procedures, OCX
OCX examples
i-ocx1.w *570*
ON statement
ANYWHERE option *286*, *290*
changing key function *282*
OpenEdge database functions *41*
OpenEdge file, colors and fonts *326*
OpenEdge session ID
defined *105*
managing *141*
setting *141*
ORDINAL option, PROCEDURE statement *487*
OS-DIR option, INPUT FROM statement *315*
OS-GETENV function *315*
_osprint.p *304*
Outer joins
See also Left outer join
full outer *68*
right outer *68*
See also Left outer join
Output
changing the destination *298*
devices *301*
files *301*
multiple destinations *302*, *309*
printers *299*
PUT statement *320*

Output *(continued)*
streams *298*
terminal *301*
Windows clipboard *301*
working within procedures *297*
OUTPUT CLOSE statement *302*, *309*
Output destinations *359*
OUTPUT mode option
method parameters *532*
OUTPUT STREAM statement *309*
OUTPUT THROUGH statement
code page conversion *317*
OUTPUT TO statement
code page conversion *317*
PAGED option *301*
Output, redirecting *359*

P

PAGE-BOTTOM option *365*
PAGE-NUMBER function *364*
PAGED option, OUTPUT TO statement *301*
Parameter files
different systems *36*
invoking *36*
Parameter File (-pf) startup parameter *36*
Parameters
definitions for shared library routines *487*
OCX event procedure
data type conversions *609*
mode option considerations *567*
passing
COM object methods *531*
named *533*
passing to DLL RUN statements *499*
PARTIAL-KEY option
SYSTEM-HELP statement *457*
PASCAL option, PROCEDURE statement *487*
Password
ENCODE function *102*
encryption *103*
Paste operations using clipboard *441*
Permissions table
adding records *159*
security *163*
using the Data Dictionary to create *159*
PERSISTENT option, PROCEDURE statement *487*
PFCOLOR
using *330*
widget assignment *327*
widget states *330*
Physical Database Name (-db) startup parameter, connection parameter *36*
Pointers, method parameters *532*
Policy program API *661*
POSITION MAXIMIZE option
SYSTEM-HELP statement *457*
POSITION X Y WIDTH HEIGHT option
SYSTEM-HELP statement *457*
Positioning ActiveX controls *558*
Preparing input files *318*
Preselect lists, See Results lists
Preselected queries *54*
Presorted queries *54*
Printers
redirecting output *299*
setting up in Windows *299*, *301*
Printing
_osprint.p *304*
converting widgets *303*
solutions *304*
text files *304*
PROCEDURE statement for DLLs
C, Pascal, and standard conventions *487*
options *487*
routines declared by number *487*
PROCEDURE statement for shared libraries
parameter data types *490*
Procedures
changing output destination *298*
changing the input source *306*
continuing processing *286*
monitoring keystrokes *264*
sharing streams *311*
using for activity authorization *160*
Processes, input and output streams *314*
PRODSP dispatch routine
defined *377*
hlprodsp.c source file *378*
mandatory declaration *378*
mapping C functions *378*
mapping routine identifiers *378*
modifying the prototype file *378*
Programming, multi-database, techniques, and considerations *91*
PROMPT-FOR statement *47*, *452*
Properties
ActiveX control access *560*
COM object, See Properties and methods, COM object
control-frame COM object *554*
Properties and methods, COM object
accessing *525*
data type conversions *609*
default *534*
locating in viewer *540*
mode and type options *531*
restrictions *533*
syntax
component handle expression *527*
data type options *530*

Properties and methods, COM object *(continued)*
syntax *(continued)*
method name reference *527*
method reference *526*
property name reference *529*
property reference *526*
Property references, COM object
restrictions *533*
_prostar.p procedure *135*
PROTERMCAP file, colors and fonts *326*
PSTimer control (OCX) *578*
PUT statement *320*, *368*
PUT-BYTE statement *90*
PUT-datatype statements *416*
PUT-KEY-VALUE statement
example *340*

Q

Queries
See also Fetching records
field lists *59*, *62*
results lists *54*
See also Fetching records
QUIT option
SYSTEM-HELP statement *457*
Quitting help from application *464*
QUOTER utility
example procedure *701*
interactive *701*
options
columns in fields *701*
field delimiter *701*

R

RAW data type *90*
Raw database events
auditing *246*
Raw disk I/O, Unix systems *396*
RAW statement *90*
Raw terminal I/O mode
Unix system recovery *398*
Read CDC policies *654*
Read methods *648–649*
Reading multiple tables, See Joins
RECID data type
See also ROWID data type
converting to ROWID *54*
DBRESTRICTIONS function *53*
See also ROWID data type
Record buffers *47*
Record phrase
field lists *59*
Records
adding *49*
Records *(continued)*
adding to a permissions table *159*
deleting *49*
fetching *50*
Redirecting I/O
input streams *306–307*
output streams *298*, *302*
Registry
changing resources *339*
colors and fonts *326*
Registry flags, Automation objects *544*
RELEASE OBJECT statement
ActiveX controls *564*
Automation objects *548*
COM objects, in general *536*
RELEASE statement *47*
Releasing COM objects *536*
REMOVE-EVENTS-PROCEDURE method *569*
Reporting
audit *255*
Reports
aggregate values *352*
control blocks *345*
editors *351*
multiple destinations *362*
multiple-table *355*
output statements *345*
printing *362*, *372*
sending to other devices *359*
text widgets *345*
viewing on screen *347*
REPOSITION statement
ROWID *53*
Required: add at least one index entry *666*
Resource, definitions, changing *339*
Results lists
navigating *56*
Retrieving records, See Fetching records
Return Codes, DDE *512*
RETURN PARAMETER option
parameter definitions *487*
RUN parameter lists *499*
RGB-VALUE function
ActiveX controls *329*
RGB-VALUE function, COM objects *535*
ROLES attribute
using with CAN-DO function *156*
ROWID data type
converting from RECID *54*
CREATE statement *53*
DataServer
portability *54*
value construction *54*
versions *54*
views *54*
INSERT statement *53*

ROWID data type *(continued)*
RECID *52*
REPOSITION statement *53*
returning record values *53*
storing values *53*
value scope *54*
ROWID function *53*
Run mode, ActiveX control
requirements *431*
RUN statement for DLLs
options *499*
parameter data types *499*
Run-time connection, methods list *40*
Run-time OCX properties *556*
Run-time authorization *97*, *155*

S

SAVE CACHE statement *94*
Save methods *649–650*
Schema authorization
defined *97*
Schema Cache File (-cache) startup parameter *96*
Schema cache files
building *94*
creating
example *94*
DataServers *94*
described *93*
startup *96*
Schema definition *450*
Screen
buffers *47*
Screen display, HLC library functions *673*
SCREEN-IO option *303*
Scrolling queries *54*
SEARCH function *302*
Search list, OCX event procedures *569*
Secure database connections *36*
Security
activity authorization procedures *160*
application *97*
auditing
defined *97*
authentication *97*
authorization, activities-based *159*
authorizing user IDs at run-time *155*
compile-time authorization
defined *97*
creating, permissions table *159*
cryptography
defined *97*
permissions table *163*
run-time authorization
defined *97*
Security *(continued)*
schema authorization
defined *97*
using secure database connections *36*
Security domains *99*
Security tokens *99*
SECURITY-POLICY system handle
application trusted domain registry *124*
cryptography *176*
SELECT mouse button *284*
SELF system handle, ActiveX control events *568*
Sequence definitions
creating and maintaining *85*
storage *85*
Sequence statements and functions *85*
_Sequence table *85*
Sequences
accessing values *85*
attributes *82*
control tables
performance *84*
storage limits *85*
trade-offs *83*
transaction handling *84*
defined *81*
dumping and loading *85*
fields *83*
security *85*
storage *85*
transactions *84*
WHERE clause *89*
Servers, Automation
accessing *544*
defined *543*
SET statement *47*, *452*
SET-APPL-CONTEXT method
usage *251*
SET-CONTENTS option
SYSTEM-HELP statement *457*
SET-DB-CLIENT function *135*
SET-SIZE statement
freeing DLL-allocated memory *497*
freeing memory *415*
SETUSERID function *135*
Shared buffer access, HLC library functions *672*
Shared libraries
data types *490*
loading and unloading *501*
RAW data type *496*
routine execution *498*
structure parameters *496*
using *484*
Shared library (UNIX), parameter definition *487*
Shared library structures, incorrect initialization *497*
Shared library, defined *483*
Shared object, defined *483*

Shared queries, field lists 60
Shared Variable Access, HLC library functions 671
Single-item transfers, clipboard
defined 437
example 443
implementing 441
techniques 439
Sizing ActiveX controls 558
Socket programming
examples 599
overview 432
Secure Sockets Layer (TLS)
configuring servers and clients 595
connecting TLS clients 597
enabling TLS server connections 596
Transport Layer Security (TLS)
socket operations 598
Sockets
See also Socket programming
ABL objects 434
connection model 433
defined 432
event model 434
programming requirements 435
See also Socket programming
Startup parameters
connection 36
Field List Disable (-fldisable) 62
grouping 36
multi-database connection 36
Number of Databases (-h) 36
Physical Database Name (-db) 36
Schema Cache File (-cache) 96
Startup Procedure (-p) 41
Trigger Location (-trig) 89
Startup procedure
specifying 41
Startup Procedure (-p) startup parameter 41
State-free identity management 151
Stateless identity management 149
Statements
data handling 47
Static data 391
Status area 449
STATUS-AREA attribute 449
STDCALL option, PROCEDURE statement 487
Stream I/O
converting widgets 303
STREAM-IO option 350
Streams
assignment 298
closing 302
default 298
defining additional 309
establishing 309
input 298, 309
Streams (continued)
opening 309
output 298, 309
redirecting 307
See also OUTPUT statements
redirecting input, See OUTPUT statements
shared 311
unnamed 298
Streams, two-way 314
String conversion, ROWID data type 53
STRING function, ROWID 53
SUBSTRING function
QUOTER utility 701
Symmetric cryptography 181
See also Cryptography
Syntax for COM objects 525
System clipboard
CLIPBOARD handle, See CLIPBOARD system handle
data transfer modes 437
defined 423, 437
enabling and disabling operations 440
example procedure
multiple-item transfer 447
single-item transfer 443
FOCUS handle 439
multiple-item transfers
data-based 446
defined 437
widget-based 446
operations
copy 442
cut 441
defined 437
paste 441
single-item transfer
defined 437
example 443
implementing 441
techniques 439
System handles, ActiveX control events 568
SYSTEM-DIALOG COLOR statement
example 333
SYSTEM-DIALOG FONT statement
example 335
SYSTEM-DIALOG PRINTER-SETUP statement 299, 301
SYSTEM-HELP statement
QUIT parameter 464

T

Tab order of ActiveX controls
design time 559
run time 563
Table policies API 647–650

Table policy *657*
Table policy API *646–647*, *649*
Table policy interface properties *657*
Tables
  audit policy *256*
  joining, See Joins
  reading multiple *63*
  word break *71*
Tables, audit data *257*
Temporary tables
  joins *64*
Terminal I/O, Unix Systems *396*
Terminating sequences, resetting *89*
TEXT option
  SYSTEM-HELP statement *457*
Tilde (~), input files *321*
Timer services
  defined *394*
  HLC library functions *673*
TLS, See Transport Layer Security (TLS)
TO-ROWID function *53*
ToolTips
  defining in the AppBuilder *455*
Top-level Automation objects *543*
Topic IDs *456*
Transactions *395*
Transport Layer Security (TLS)
  database connections *36*
  socket connections *595*
Trigger Location (-trig) startup parameter *89*
Triggers
  help *460*, *462*
Two-way streams *314*
TYPE attribute *439*
Type Libraries, defined *531*
Type options
  purpose *531*
  using *532*
Type specifiers
  COM object syntax *530*
  purpose *531*
  using *531*

U

Unix
  HLC applications *396*
  shared libraries
    and HLC *376*
UNIX
  shared libraries
    and ABL *425*
    defined *424*
UNIX, processes as input and output streams *314*
UNLOAD statement
  using *341*
Unnamed streams *298*
Update CDC policies *655*
Update methods *649–650*
UPDATE statement *47*, *452*
USE statement
  using *341*
User count, exceeding *36*
User credentials *99*
User identities *104*
  See also Identity management
User identity, See Identity
User identity management mechanisms *100*
User IDs
  authenticated and default *104*
  authorizing at run time *155*
  CAN-DO validation list *156*
User interrupt handling *395*
USERID function *156*
User-interface triggers
  instead of EDITING blocks *290*

V

Validating component handles *534*
Validation
  field-level *318*
VALUE attribute
  multiple-item transfers
    example *447*
  single-item transfers
    example *443*
VALUE option
  OUTPUT TO statement *302*
Version, of OpenEdge, determining the current *277*
VIEW-AS phrase *454*
Visual colors, Windows color dialog *333*

W

Widget handles, compared to component handles *524*
Widget hierarchy *463*
Widgets
  converting for stream I/O *303*
Window status area *449*
WINDOW-NAME option
  SYSTEM-HELP statement *457*
Windows
  clipboard output *301*
  color
    Palette Manager *331*
  printer setup *299*, *301*
Windows NT named pipe *476*
WinExec() DLL function
  code example *512*
  DDE *512*

Word break tables
internationalization 71
Word indexing 71
Word-break, rules file
applying to database 77
Word-break, rules file (continued)
compiling 76
Work tables
joins 64