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.

Actional, Apama, Apama (and Design), Artix, Business Empowerment, DataDirect (and design), DataDirect Connect, DataDirect Connect64, DataDirect Technologies, DataDirect XML Converters, DataDirect XQuery, DataXtend, Dynamic Routing Architecture, EdgeXtend, Empowerment Center, Fathom, IntelliStream, IONA, IONA (and design), Making Software Work Together, Mindreef, ObjectStore, OpenEdge, Orbix, PeerDirect, POSSENET, Powered by Progress, PowerTier, Progress, Progress DataXtend, Progress Dynamics, Progress Business Empowerment, Progress Empowerment Center, Progress Empowerment Program, Progress OpenEdge, Progress Profiles, Progress Results, Progress Software Developers Network, Progress Sonic, ProVision, PS Select, Sequelink, Shadow, SOAPscope, SOAPStation, Sonic, Sonic ESB, SonicMQ, Sonic Orchestration Server, SonicSynergy, SpeedScript, Stylus Studio, Technical Empowerment, WebSpeed, Xcalia (and design), and Your Software, Our Technology–Experience the Connection are registered trademarks of Progress Software Corporation or one of its affiliates in the U.S. and/or other countries.

AccelEvent, Apama Dashboard Studio, Apama Event Manager, Apama Event Modeler, Apama Event Store, Apama Risk Firewall, AppsAlive, AppServer, ASPen, ASP-in-a-Box, BusinessEdge, Business Making Progress, Cache-Forward, DataDirect Spy, DataDirect SupportLink, Fuse, Fuse Mediation Router, Fuse Message Broker, Fuse Services Framework, Future Proof, GVAC, High Performance Integration, ObjectStore Inspector, ObjectStore Performance Expert, OpenAccess, Orbusc, Pantero, POSSE, ProDataSet, Progress ESP Event Manager, Progress ESP Event Modeler, Progress Event Engine, Progress RFID, Progress Software Business Making Progress, PSE Pro, SectorAlliance, SeeThinkAct, Shadow z/Services, Shadow z/Direct, Shadow z/Events, Shadow z/Presentation, Shadow Studio, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, Sonic Business Integration Suite, Sonic Process Manager, Sonic Collaboration Server, Sonic Continuous Availability Architecture, Sonic Database Service, Sonic Workbench, Sonic XML Server, StormGlass, The Brains Behind BAM, WebClient, Who Makes Progress, and Your World. Your SOA. are trademarks or service marks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and other countries. Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. Any other trademarks contained herein are the property of their respective owners.

**Third party acknowledgements** — See the “Third party acknowledgements” section on page Preface–12.

December 2009

Last updated with new content: Release 10.2B

Product Code: 4496; R10.2B

For the latest documentation updates see OpenEdge Product Documentation on PSDN (http://communities.progress.com/pcom/docs/DOC-16074).
2. Application Security .......................................................... 2–1
Types of application security .............................................. 2–2
Authenticating user identities .............................................. 2–3
Specifying OpenEdge internal user IDs and passwords .......... 2–3
Authenticating OpenEdge internal user IDs and passwords .. 2–4
Using externally-authenticated user IDs and passwords ...... 2–6
Managing identities ........................................................... 2–7
ABL for managing user identities ....................................... 2–8
Client-principal objects .................................................... 2–9
Managing client login sessions ....................................... 2–16
Managing database connection IDs .................................. 2–19
Managing OpenEdge session IDs ...................................... 2–23
Managing application user IDs in n-tier applications ....... 2–28
Managing auditing IDs ..................................................... 2–36
Creating and managing unique object identities .............. 2–36
Using secure database connections ................................. 2–37
Authorizing access to procedures and database resources .... 2–39
Authorizing users from within a given procedure .......... 2–39
Defining activities-based user authorization ...................... 2–42
Authorizing user access to tables and fields .................... 2–49
Using cryptography to secure data ..................................... 2–51
Creating and maintaining a cryptography policy .......... 2–52
Generating encryption keys .............................................. 2–54
6. **Colors and Fonts** .................................................. 6–1
   Making colors and fonts available to an application .................. 6–2
   OpenEdge default colors .............................................. 6–3
   OpenEdge default fonts ................................................ 6–4
   Assigning colors and fonts to a widget .............................. 6–5
   Assigning colors and fonts to ActiveX Automation objects and ActiveX controls ....................... 6–7
   Color and font inheritance ............................................ 6–8
   Overriding default color inheritance ................................ 6–8
   Color in character interfaces ......................................... 6–9
   Widget states and color ............................................... 6–9
   Color specification .................................................... 6–9
   Colors in windows in graphical interfaces .......................... 6–11
   Systems with the Palette Manager .................................. 6–11
   Systems without the Palette Manager ............................... 6–11
   Managing colors and fonts in graphical applications ............. 6–12
   Allowing the user to change colors and fonts ....................... 6–13
   Establishing dynamic colors ......................................... 6–13
   Color dialog box ..................................................... 6–13
   Color dialog box example ............................................ 6–15
   Saving a modified color .............................................. 6–16
   Font dialog box ....................................................... 6–16
   Font dialog box example .............................................. 6–17
   Saving a modified font ................................................ 6–18
   Accessing the current color and font tables ....................... 6–19
   COLOR–TABLE handle ............................................... 6–19
   FONT–TABLE handle .................................................. 6–20
   Retrieving and changing color and font definitions .............. 6–21
   Changing resource definitions ....................................... 6–21
   Using GET–KEY–VALUE and PUT–KEY–VALUE ..................... 6–21
   Managing application environments .................................. 6–23
   Understanding environment management ............................ 6–23
   Using environment management statements ......................... 6–23
   Managing multiple environments .................................... 6–24
   Managing color display limitations .................................. 6–26

7. **Creating Reports** .................................................. 7–1
   Report basics .......................................................... 7–2
   Down frames ........................................................... 7–2
   Text widgets ........................................................... 7–3
   Control widgets and output statements .............................. 7–3
   Basic report demonstration .......................................... 7–4
Part III External Program Interfaces

8. Introduction to External Program Interfaces ......................................................... 8–1
   Using MEMPTR to reference external data ............................................................... 8–2
      Comparing MEMPTR and RAW data types .......................................................... 8–3
      Initializing and uninitializing MEMPTR variables ............................................... 8–3
      Reading and writing data ...................................................................................... 8–5
      Retrieving and storing pointers ............................................................................. 8–10
      Setting byte order ................................................................................................. 8–11
   System clipboard ....................................................................................................... 8–12
      The system clipboard and ABL .............................................................................. 8–12
      Requirements for using the CLIPBOARD handle ................................................. 8–12
   Named pipes .............................................................................................................. 8–13
      Named pipes and ABL ............................................................................................. 8–13
      Requirements for using named pipes ..................................................................... 8–13
   UNIX shared library and Windows DLL support ...................................................... 8–14
      Shared libraries and ABL ....................................................................................... 8–14
      Requirements for using shared libraries ............................................................... 8–15
   Windows Dynamic Data Exchange ........................................................................... 8–16
      DDE and ABL .......................................................................................................... 8–16
      Requirements for using DDE .................................................................................. 8–17
   COM objects: Automation objects and ActiveX controls .......................................... 8–18
      COM objects supported in ABL .............................................................................. 8–18
      Support for COM object properties and methods ............................................... 8–21
      Support for Automation object events ................................................................... 8–21
      Support for ActiveX control events ....................................................................... 8–21
      COM object sources .............................................................................................. 8–22
      Requirements for using Automation objects .......................................................... 8–22
      Requirements for using ActiveX controls .............................................................. 8–22
      Programming requirements ................................................................................... 8–24
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sockets</td>
<td>8–25</td>
</tr>
<tr>
<td>Reasons to use sockets</td>
<td>8–25</td>
</tr>
<tr>
<td>Connection model</td>
<td>8–26</td>
</tr>
<tr>
<td>Server socket and socket objects</td>
<td>8–27</td>
</tr>
<tr>
<td>ABL socket event model</td>
<td>8–27</td>
</tr>
<tr>
<td>Programming requirements</td>
<td>8–29</td>
</tr>
<tr>
<td><strong>9. System Clipboard</strong></td>
<td>9–1</td>
</tr>
<tr>
<td>CLIPBOARD system handle</td>
<td>9–2</td>
</tr>
<tr>
<td>AVAILABLE-FORMATS Attribute</td>
<td>9–2</td>
</tr>
<tr>
<td>ITEMS-PER-ROW Attribute</td>
<td>9–3</td>
</tr>
<tr>
<td>MULTIPLE Attribute</td>
<td>9–3</td>
</tr>
<tr>
<td>NUM-FORMATS Attribute</td>
<td>9–3</td>
</tr>
<tr>
<td>TYPE Attribute</td>
<td>9–3</td>
</tr>
<tr>
<td>VALUE attribute</td>
<td>9–3</td>
</tr>
<tr>
<td>Single-item data transfers</td>
<td>9–4</td>
</tr>
<tr>
<td>Enabling and disabling clipboard operations</td>
<td>9–4</td>
</tr>
<tr>
<td>Implementing single-item transfers</td>
<td>9–6</td>
</tr>
<tr>
<td>Single-item transfer example</td>
<td>9–8</td>
</tr>
<tr>
<td>Multiple-item data transfers</td>
<td>9–12</td>
</tr>
<tr>
<td>Widget-based transfers</td>
<td>9–12</td>
</tr>
<tr>
<td>Data-based transfers</td>
<td>9–13</td>
</tr>
<tr>
<td>Multiple-item transfer example</td>
<td>9–14</td>
</tr>
<tr>
<td><strong>10. Providing Help for OpenEdge Applications</strong></td>
<td>10–1</td>
</tr>
<tr>
<td>Field-level online help</td>
<td>10–2</td>
</tr>
<tr>
<td>Status messages</td>
<td>10–2</td>
</tr>
<tr>
<td>ToolTips</td>
<td>10–6</td>
</tr>
<tr>
<td>Online help systems</td>
<td>10–9</td>
</tr>
<tr>
<td>The SYSTEM-HELP statement</td>
<td>10–11</td>
</tr>
<tr>
<td>Accessing online help from the menu bar</td>
<td>10–17</td>
</tr>
<tr>
<td>Calling online help with a Help button</td>
<td>10–19</td>
</tr>
<tr>
<td>Accessing online help with the help key</td>
<td>10–20</td>
</tr>
<tr>
<td>Quitting help when exiting the application</td>
<td>10–21</td>
</tr>
<tr>
<td><strong>11. Named Pipes</strong></td>
<td>11–1</td>
</tr>
<tr>
<td>Overview of named pipes with ABL</td>
<td>11–2</td>
</tr>
<tr>
<td>Operational characteristics of named pipes</td>
<td>11–3</td>
</tr>
<tr>
<td>Advantages and disadvantages of named pipes</td>
<td>11–4</td>
</tr>
<tr>
<td>UNIX named pipes</td>
<td>11–5</td>
</tr>
<tr>
<td>Creating a named pipe</td>
<td>11–5</td>
</tr>
<tr>
<td>Deleting a named pipe</td>
<td>11–6</td>
</tr>
<tr>
<td>Accessing a named pipe within ABL</td>
<td>11–6</td>
</tr>
<tr>
<td>UNIX named pipe examples</td>
<td>11–7</td>
</tr>
<tr>
<td>Windows named pipes</td>
<td>11–14</td>
</tr>
<tr>
<td>Accessing Windows named pipes</td>
<td>11–14</td>
</tr>
<tr>
<td>Linking OpenEdge and non-OpenEdge processes using Windows named pipes</td>
<td>11–14</td>
</tr>
<tr>
<td>Building and running the sample Windows named pipes application</td>
<td>11–15</td>
</tr>
<tr>
<td>Coding the ABL program</td>
<td>11–15</td>
</tr>
<tr>
<td>Coding the C program</td>
<td>11–18</td>
</tr>
<tr>
<td>Running the application</td>
<td>11–20</td>
</tr>
</tbody>
</table>
12. **Shared Library and DLL Support** ................................................. 12–1
   Using shared libraries ............................................................... 12–2
   Accessing a shared library entry point ........................................ 12–5
   Declaring a shared library for static access ................................. 12–5
   Declaring a shared library for dynamic access .............................. 12–6
   Passing parameters to a shared library routine ................................ 12–9
   Shared library parameter data types .......................................... 12–9
   Passing arrays as parameters ..................................................... 12–11
   Passing the Unknown value and NULL values ............................... 12–15
   Using structure parameters ....................................................... 12–16
   Using MEMPTR variables as parameters ...................................... 12–16
   DLL routines and ABL widgets (Windows only) .............................. 12–18
   Executing a shared library routine .............................................. 12–19
   Executing a shared library routine for static access ....................... 12–19
   Executing a shared library routine for dynamic access .................... 12–19
   Parameter data types ............................................................... 12–20
   INT64 parameter conversions ...................................................... 12–20
   Loading and unloading shared libraries ...................................... 12–22
   Code samples ........................................................................... 12–23
   Examples of static access to a shared library ............................... 12–23
   Examples of dynamic access to a shared library ............................ 12–26

13. **Windows Dynamic Data Exchange** ............................................. 13–1
   Introduction to DDE ..................................................................... 13–2
   Using the DDE protocol ................................................................ 13–3
   The course of a DDE conversation ............................................... 13–3
   ABL statements for DDE conversations ........................................ 13–4
   Structuring a DDE conversation in ABL ........................................ 13–5
   Conversation hierarchy .................................................................. 13–5
   Defining conversation endpoints—DDE frames ............................... 13–7
   DDE-ERROR .............................................................................. 13–8
   Other DDE attributes ................................................................. 13–8
   Opening DDE conversations .......................................................... 13–9
   Preparing the server application .................................................... 13–9
   Defining DDE frames .................................................................... 13–10
   Initiating a conversation ................................................................ 13–11
   Exchanging data in conversations ................................................ 13–12
   Demand-driven exchanges ............................................................ 13–12
   Event-driven exchanges ................................................................ 13–13
   Closing DDE conversations .......................................................... 13–16
   DDE example .............................................................................. 13–17

14. **Using COM Objects in ABL** .......................................................... 14–1
   How COM objects differ from ABL widgets .................................... 14–2
   Functionality ................................................................................ 14–2
   ABL mechanisms .......................................................................... 14–2
   Obtaining access to COM objects ................................................ 14–4
   Accessing COM object properties and methods .............................. 14–5
   Property and method syntax ........................................................ 14–5
   Specifying options for properties and method parameters ............... 14–10
   Restrictions on property and method references ............................ 14–13
   Managing COM objects in an application ...................................... 14–14
   Validating component handles ...................................................... 14–14
   Managing COM object font and color settings ............................... 14–14
Navigating ActiveX collections .................................... 14–15
Managing COM object resources .................................. 14–16
Error handling ......................................................... 14–17
Locating COM object information on your system .............. 14–18
Using the COM Object Viewer ..................................... 14–18
Accessing Type Libraries ........................................... 14–18
Locating objects in the Viewer ....................................... 14–19
Viewing methods, properties, and events ......................... 14–21

15. ActiveX Automation Support ..................................... 15–1
Requirements for doing Automation .................................. 15–2
Accessing Automation Servers ....................................... 15–3
Option 1: Instantiate Automation object by name ............... 15–4
Option 2: Connect to top-level named Automation object ....... 15–5
Option 3: Connect to or instantiate a named Automation object and file ............................................. 15–6
Option 4: Connect to or instantiate implied Automation object and file ............................................. 15–7
Managing Automation objects ....................................... 15–8
Automation event support ............................................. 15–9
Example Automation applications ................................. 15–10

16. ActiveX Control Support ............................................ 16–1
How ABL supports ActiveX controls ............................... 16–2
An example ActiveX control in ABL ............................... 16–2
How ABL encapsulates an ActiveX control ....................... 16–3
Control-frame attributes and properties ......................... 16–4
Additional control functionality .................................... 16–4
Special considerations for extended controls .................... 16–6
ActiveX control restrictions ........................................ 16–6
Creating a control instance in ABL ............................... 16–7
Understanding design time and runtime properties ............ 16–7
Setting design time properties ...................................... 16–8
Setting the ActiveX control name .................................. 16–8
Setting the control-frame name ..................................... 16–9
Orienting a control in the user interface at design time ........ 16–10
Setting control location .............................................. 16–10
Setting control height and width ................................... 16–10
Setting control tab order ............................................. 16–11
Defining invisible controls .......................................... 16–11
Accessing ActiveX controls at runtime ......................... 16–12
Instantiating the control ............................................ 16–12
Accessing the control handle ........................................ 16–13
Managing ActiveX controls at runtime ......................... 16–16
Managing tab order and Z order .................................... 16–16
Working with ABL key functions ................................... 16–16
Setting graphical properties of an ActiveX control ............ 16–17
Releasing control resources ......................................... 16–17
Handling events ....................................................... 16–19
Handling control-frame widget events ............................ 16–19
Handling ActiveX control events ................................... 16–19
Managing external procedure files ................................. 16–24
Programming ActiveX controls in the AppBuilder ............. 16–25
Creating data definitions for an ActiveX control ............... 16–25
Instantiating the control ............................................ 16–28
Initializing the control .............................................. 16–31
17. Sockets ........................................... 17–1
ABL for programming sockets ...................................... 17–2
Overview of tasks to implement ABL sockets ...................... 17–7
Implementing an ABL socket server .............................. 17–9
  Listening and responding to connection requests ................. 17–9
  Managing the server socket ...................................... 17–10
Implementing an ABL socket client ................................ 17–11
Read, writing, and managing sockets on clients and servers .... 17–12
  Defining and initializing MEMPTR variables ...................... 17–12
  Detecting data on a socket ....................................... 17–12
  Reading data on a socket ....................................... 17–14
  Writing data on a socket ....................................... 17–16
  Marshalling and unmarshalling data for socket I/O .............. 17–16
  Managing sockets and their connections ...................... 17–17
Implementing ABL socket security .................................. 17–18
  Configuring SSL servers and clients .......................... 17–19
  Enabling SSL server connections ................................ 17–20
  Connecting an SSL client to an SSL server .................... 17–21
  Invoking socket operations for SSL sessions .................. 17–21
Examples using ABL sockets ........................................ 17–23

18. Host Language Call Interface ................................. 18–1
HLC and OpenEdge .............................................. 18–2
  Requirements for using HLC ..................................... 18–2
Using HLC ...................................................... 18–3
  Overview of HLC ............................................... 18–4
    Using the CALL statement .................................. 18–5
    Mapping routine identifiers using PRODSP() ................. 18–6
HLC files and directories .......................................... 18–9
Writing C functions ............................................ 18–10
  Top-level C function declaration ................................ 18–11
  Returning error codes from a top-level function .............. 18–11
  Naming C functions ........................................... 18–12
  C function portability ......................................... 18–12
Avoiding common HLC errors ....................................... 18–13
Memory allocation .................................................. 18–14
Data size ......................................................... 18–15
Using HLC library functions ........................................ 18–16
  Accessing ABL data ........................................... 18–16
  Data type conversion ........................................... 18–18
  Calling an HLC library function .............................. 18–19
  Timer services ............................................... 18–19
  User interrupt handling ....................................... 18–20
  Using a library function that writes to a shared buffer ....... 18–20
  Passing error codes back to ABL ............................. 18–21
Building an HLC executable ........................................ 18–22
HLC applications on UNIX systems ................................ 18–23
  Handling raw disk I/O ......................................... 18–23
Part IV Appendices

A. COM Object Data Type Mapping .............................................. A–1
   Data type conversion strategy .............................................. A–2
      Converting COM to ABL data types ................................. A–2
      Converting ABL to COM data types ................................. A–2
      General conversion features and limitations .................. A–2
   Conversions from COM to ABL data types ......................... A–3
   Conversions from ABL to COM data types ......................... A–4
      ABL to COM data type features and limitations ............. A–5
   Alternate COM data type names ...................................... A–7

B. Audit Policy Maintenance APIs ............................................. B–1
   API overview ................................................................. B–2
      Reference to API procedure files ................................. B–3
      Using the generic utility and caching APIs with an AppServer ... B–4
   Generic utility API ......................................................... B–5
      cancel-import-from-xml procedure ............................... B–5
      check-conflicts procedure ......................................... B–6
      export-audit-events procedure .................................... B–6
      export-cached-audit-events procedure ......................... B–7
      fill-audit-dataset procedure ...................................... B–7
      getAuditEvents procedure ......................................... B–8
      get-complete-dbname-list procedure ............................ B–8
      get-DB-Name function ............................................... B–9
      get-dbname-list procedure ......................................... B–9
      get-policies-merge procedure .................................... B–10
      has-DB-option function ............................................. B–10
      isDBOnAppServer function ........................................... B–11
      override-policies-from-xml procedure ......................... B–12
      policies-dataset-read-xml procedure ............................ B–12
      policies-dataset-write-xml procedure ........................... B–13
      setAppServerHandle procedure .................................... B–13
      updateAuditEvent procedure ....................................... B–13
      update-auditing-policies procedure .............................. B–14
   Caching API ................................................................. B–16
      changeAuditDatabase procedure .................................. B–16
      export-cached-audit-events procedure ......................... B–18
      export-policies-to-xml procedure ................................ B–19
      get-audit-events procedure ........................................ B–19
      get-conflict-info procedure ..................................... B–20
      getDataModified procedure ...................................... B–20
      get-merge-info procedure ........................................ B–20
      getNextAppLevelEventID function .............................. B–20
      hasApplEvents function ............................................. B–20
      import-audit-events procedure ...................................... B–21
| Table 1–1: ABL elements for managing user identities | 2–8 |
| Table 1–2: ABL database functions | 1–7 |
| Table 1–3: Connection failure behavior | 1–5 |
| Table 1–4: Results lists for specific query types | 1–23 |
| Table 1–5: Is the dot a word delimiter? | 1–43 |
| Table 1–6: Word delimiter attributes | 1–43 |
| Table 1–7: Comparison of sequences and control tables | 1–55 |
| Table 1–8: Sequence statements and functions | 1–58 |
| Table 2–1: ABL elements for managing user identities | 2–8 |
| Table 2–2: Client-principal object attributes you can read-only | 2–11 |
| Table 2–3: Client-principal object attributes you can set | 2–11 |
| Table 2–4: Client-principal object methods | 2–13 |
| Table 2–5: Methods for building an application domain registry | 2–24 |
| Table 2–6: Identity management for n-tier applications | 2–29 |
| Table 2–7: Unique value functions | 2–36 |
| Table 2–8: Data encoding/decoding functions | 2–37 |
| Table 2–9: SSL database connection parameters | 2–38 |
| Table 2–10: Values to use for ID lists | 2–41 |
| Table 2–11: CAN-* attributes for testing run-time data access permissions | 2–49 |
| Table 2–12: SECURITY-POLICY system handle attributes | 2–52 |
| Table 2–13: Cryptographic key generation functions | 2–55 |
| Table 2–14: ENCRYPT/DECRYPT function parameters | 2–57 |
| Table 2–1: ABL elements for managing user identities | 2–8 |
| Table 2–2: Client-principal object attributes you can read-only | 2–11 |
| Table 2–3: Client-principal object attributes you can set | 2–11 |
| Table 2–4: Client-principal object methods | 2–13 |
| Table 2–5: Methods for building an application domain registry | 2–24 |
| Table 2–6: Identity management for n-tier applications | 2–29 |
| Table 2–7: Unique value functions | 2–36 |
| Table 2–8: Data encoding/decoding functions | 2–37 |
| Table 2–9: SSL database connection parameters | 2–38 |
| Table 2–10: Values to use for ID lists | 2–41 |
| Table 2–11: CAN-* attributes for testing run-time data access permissions | 2–49 |
| Table 2–12: SECURITY-POLICY system handle attributes | 2–52 |
| Table 2–13: Cryptographic key generation functions | 2–55 |
| Table 2–14: ENCRYPT/DECRYPT function parameters | 2–57 |
| Table 3–1: Auditing privilege capabilities | 3–7 |
| Table 3–2: Methods for generating application events | 3–9 |
| Table 3–3: ABL elements for auditing client login session context | 3–13 |
| Table 4–1: ABL key input precedence | 4–3 |
| Table 4–2: Preferred key codes and key labels | 4–6 |
| Table 4–3: Alternate key codes and key labels | 4–15 |
| Table 4–4: ABL key functions | 4–16 |
| Table 4–5: Actions you assign to keys | 4–21 |
| Table 4–6: Mouse buttons in Windows | 4–22 |
| Table 5–1: Printing options for Windows | 5–10 |
| Table 5–2: Using streams | 5–22 |
| Table 6–1: OpenEdge default colors | 6–3 |
| Table 6–2: Colors used in character interfaces | 6–9 |
| Table 7–1: Aggregate options | 7–12 |
| Table 7–2: OUTPUT statement options | 7–20 |
| Table 8–1: PUT-datatype statements | 8–6 |
| Table 8–2: GET-datatype functions | 8–7 |
| Table 8–3: Copying between basic ABL data types and MEMPTR | 8–8 |
| Table 9–1: CLIPBOARD handle attributes | 9–2 |
| Table 10–1: Using C and ABL to access Windows named pipes | 11–14 |
| Table 11–1: Comparison of static and dynamic shared library access | 12–3 |
| Table 12–1: Comparison of static and dynamic shared library access | 12–3 |
| Table 12–2: DLL calling conventions | 12–6 |
| Table 12–3: C and DLL parameter data type compatibilities | 12–10 |
| Table 12–4: Compatibility of shared library and ABL parameters | 12–20 |
| Table 12–5: Loading and unloading shared libraries | 12–22 |
| Table 13–1: DDE statements | 13–4 |
| Table 13–2: DDE frame attributes | 13–7 |
| Table 13–3: ABL DDE errors | 13–8 |
| Table 14–1: COM objects and ABL widgets compared | 14–3 |
| Table 14–2: Data-types for COM object methods and properties | 14–9 |
| Table 15–1: Control-frame attributes and properties | 16–4 |
| Table 16–1: Extended properties | 16–5 |
| Table 16–2: Property definitions | 16–5 |
Table 16–4: Data type mappings for OCX event parameters .......................... 16–21
Table 17–1: ABL for programming sockets ........................................... 17–2
Table 17–2: Summary of socket programming tasks ................................. 17–7
Table 17–3: Socket reading modes ...................................................... 17–14
Table 17–4: Effect of read mode and timeout value on READ()'s timeout behavior ................................................................. 17–15
Table 17–5: Server socket object SSL connection parameters ................. 17–20
Table 17–6: Client socket object SSL connection parameters .................. 17–21
Table 18–1: HLC directories ............................................................... 18–9
Table 18–2: HLC filenames ................................................................. 18–9
Table A–1: Default conversions from COM data types to ABL data types ... A–3
Table A–2: Conversions from ABL to COM data types ............................ A–4
Table A–3: Alternative COM data type names ...................................... A–7
Table B–1: Audit Policy Maintenance API procedure files ....................... B–3
Table C–1: Functions that access ABL shared variables .......................... C–2
Table C–2: Functions that access ABL shared buffers ............................. C–3
Table C–3: Functions that interact with ABL displays .............................. C–4
Table C–4: Functions that test for ABL interrupts ................................. C–4
Table C–5: Functions that access ABL timer services .............................. C–4
Table D–1: QUOTER examples .......................................................... D–3
### Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1–1</td>
<td>Data movement</td>
<td>1–15</td>
</tr>
<tr>
<td>Figure 1–2</td>
<td>The primary data-handling statements</td>
<td>1–16</td>
</tr>
<tr>
<td>Figure 1–3</td>
<td>Inner joins</td>
<td>1–34</td>
</tr>
<tr>
<td>Figure 1–4</td>
<td>Left outer joins</td>
<td>1–36</td>
</tr>
<tr>
<td>Figure 1–5</td>
<td>Inner join example</td>
<td>1–38</td>
</tr>
<tr>
<td>Figure 1–6</td>
<td>Left outer join example</td>
<td>1–39</td>
</tr>
<tr>
<td>Figure 1–7</td>
<td>Database fields and sequences comparison</td>
<td>1–54</td>
</tr>
<tr>
<td>Figure 1–8</td>
<td>Sequences and overlapping transactions</td>
<td>1–56</td>
</tr>
<tr>
<td>Figure 1–9</td>
<td>Positioning database references</td>
<td>1–67</td>
</tr>
<tr>
<td>Figure 2–1</td>
<td>Sample activity permissions entries</td>
<td>2–43</td>
</tr>
<tr>
<td>Figure 2–2</td>
<td>User with permission to run a procedure</td>
<td>2–44</td>
</tr>
<tr>
<td>Figure 2–3</td>
<td>User without permission to run a procedure</td>
<td>2–44</td>
</tr>
<tr>
<td>Figure 2–4</td>
<td>Sample security record for activity permissions</td>
<td>2–47</td>
</tr>
<tr>
<td>Figure 2–5</td>
<td>Summary of run-time authorization process</td>
<td>2–48</td>
</tr>
<tr>
<td>Figure 4–1</td>
<td>Key labels, key codes, and key functions</td>
<td>4–2</td>
</tr>
<tr>
<td>Figure 4–2</td>
<td>ABL key translation functions</td>
<td>4–3</td>
</tr>
<tr>
<td>Figure 4–3</td>
<td>Key codes</td>
<td>4–5</td>
</tr>
<tr>
<td>Figure 4–4</td>
<td>Key function syntax and meaning</td>
<td>4–20</td>
</tr>
<tr>
<td>Figure 4–5</td>
<td>The i-gon1.p procedure</td>
<td>4–26</td>
</tr>
<tr>
<td>Figure 5–1</td>
<td>The unnamed streams</td>
<td>5–2</td>
</tr>
<tr>
<td>Figure 5–2</td>
<td>Redirecting the unnamed output stream</td>
<td>5–3</td>
</tr>
<tr>
<td>Figure 5–3</td>
<td>Multiple output destinations</td>
<td>5–7</td>
</tr>
<tr>
<td>Figure 5–4</td>
<td>The OUTPUT CLOSE statement</td>
<td>5–8</td>
</tr>
<tr>
<td>Figure 5–5</td>
<td>Redirecting the unnamed input stream</td>
<td>5–13</td>
</tr>
<tr>
<td>Figure 5–6</td>
<td>Multiple input sources</td>
<td>5–15</td>
</tr>
<tr>
<td>Figure 5–7</td>
<td>Multiple output streams scenario</td>
<td>5–17</td>
</tr>
<tr>
<td>Figure 6–1</td>
<td>Windows system color dialog box</td>
<td>6–13</td>
</tr>
<tr>
<td>Figure 6–2</td>
<td>Windows Font dialog box</td>
<td>6–16</td>
</tr>
<tr>
<td>Figure 6–3</td>
<td>Changing font 11 to bold 8-point Arial</td>
<td>6–18</td>
</tr>
<tr>
<td>Figure 6–2</td>
<td>ABL interactions using the system clipboard</td>
<td>8–12</td>
</tr>
<tr>
<td>Figure 6–2</td>
<td>ABL exchanging data using named pipes</td>
<td>8–13</td>
</tr>
<tr>
<td>Figure 8–2</td>
<td>OpenEdge application calling DLL functions</td>
<td>8–14</td>
</tr>
<tr>
<td>Figure 8–4</td>
<td>ABL exchanging data using DDE</td>
<td>8–16</td>
</tr>
<tr>
<td>Figure 8–5</td>
<td>ABL using ActiveX Automation</td>
<td>8–19</td>
</tr>
<tr>
<td>Figure 8–6</td>
<td>ABL interface to ActiveX Controls</td>
<td>8–20</td>
</tr>
<tr>
<td>Figure 8–7</td>
<td>ABL socket event model</td>
<td>8–28</td>
</tr>
<tr>
<td>Figure 10–1</td>
<td>Status bar message for a field-level object</td>
<td>10–2</td>
</tr>
<tr>
<td>Figure 10–2</td>
<td>Sample database field help text in the status area of a window</td>
<td>10–4</td>
</tr>
<tr>
<td>Figure 10–3</td>
<td>ToolTip example</td>
<td>10–6</td>
</tr>
<tr>
<td>Figure 10–4</td>
<td>HTML Help Viewer showing Procedure Editor Help</td>
<td>10–18</td>
</tr>
<tr>
<td>Figure 10–5</td>
<td>Help Viewer with Index tab on top</td>
<td>10–19</td>
</tr>
<tr>
<td>Figure 10–6</td>
<td>Dialog box with a Help button</td>
<td>10–19</td>
</tr>
<tr>
<td>Figure 10–7</td>
<td>Context-sensitive help for the Buffer Information dialog box</td>
<td>10–20</td>
</tr>
<tr>
<td>Figure 10–8</td>
<td>HELP Event Hierarchy</td>
<td>10–21</td>
</tr>
<tr>
<td>Figure 11–1</td>
<td>Typical named pipe scenario</td>
<td>11–2</td>
</tr>
<tr>
<td>Figure 11–2</td>
<td>Writing messages to a named pipe</td>
<td>11–3</td>
</tr>
<tr>
<td>Figure 11–3</td>
<td>Named pipes and ABL</td>
<td>11–10</td>
</tr>
<tr>
<td>Figure 13–1</td>
<td>ABL DDE conversations</td>
<td>13–6</td>
</tr>
<tr>
<td>Figure 14–1</td>
<td>Automation objects in the COM Object Viewer</td>
<td>14–19</td>
</tr>
<tr>
<td>Figure 14–2</td>
<td>ActiveX controls in the COM Object Viewer</td>
<td>14–20</td>
</tr>
<tr>
<td>Figure 15–1</td>
<td>Automation connection option 1</td>
<td>15–4</td>
</tr>
<tr>
<td>Figure 15–2</td>
<td>Automation connection option 2</td>
<td>15–5</td>
</tr>
<tr>
<td>Figure 15–3</td>
<td>Automation connection option 3</td>
<td>15–6</td>
</tr>
<tr>
<td>Figure 15–4</td>
<td>Automation connection option 4</td>
<td>15–7</td>
</tr>
<tr>
<td>Figure 16–1</td>
<td>ActiveX control example</td>
<td>16–2</td>
</tr>
</tbody>
</table>
## Procedures

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>parm3.p</td>
<td>1–4</td>
</tr>
<tr>
<td>i-topproc.p</td>
<td>1–4</td>
</tr>
<tr>
<td>i-subproc.p</td>
<td>1–4</td>
</tr>
<tr>
<td>i-infor.p</td>
<td>1–7</td>
</tr>
<tr>
<td>i-mainproc1.p</td>
<td>1–9</td>
</tr>
<tr>
<td>i-subproc1.p</td>
<td>1–9</td>
</tr>
<tr>
<td>i-subproc2.p</td>
<td>1–9</td>
</tr>
<tr>
<td>i-alias1.p</td>
<td>1–12</td>
</tr>
<tr>
<td>i-alias2.p</td>
<td>1–12</td>
</tr>
<tr>
<td>i-dispcust.p</td>
<td>1–12</td>
</tr>
<tr>
<td>i-main2.p</td>
<td>1–14</td>
</tr>
<tr>
<td>i-makebuf.p</td>
<td>1–14</td>
</tr>
<tr>
<td>i-disp.p</td>
<td>1–14</td>
</tr>
<tr>
<td>i-fldls1.p</td>
<td>1–30</td>
</tr>
<tr>
<td>i-rldls2.p</td>
<td>1–30</td>
</tr>
<tr>
<td>i-join1.p</td>
<td>1–37</td>
</tr>
<tr>
<td>i-join2.p</td>
<td>1–38</td>
</tr>
<tr>
<td>i-schcs1.p</td>
<td>1–71</td>
</tr>
<tr>
<td>i-schcs2.p</td>
<td>1–72</td>
</tr>
<tr>
<td>_login.p</td>
<td>2–5</td>
</tr>
<tr>
<td>i-passts.p</td>
<td>2–6</td>
</tr>
<tr>
<td>Login session for an internally authenticated database connection ID</td>
<td>2–20</td>
</tr>
<tr>
<td>Login session for an externally authenticated database connection ID</td>
<td>2–22</td>
</tr>
<tr>
<td>Login session for an externally authenticated OpenEdge session ID</td>
<td>2–27</td>
</tr>
<tr>
<td>Client-principal object initialization for n-tier applications</td>
<td>2–30</td>
</tr>
<tr>
<td>i-csmnu3.p</td>
<td>2–39</td>
</tr>
<tr>
<td>i-adcus3.p</td>
<td>2–40</td>
</tr>
<tr>
<td>i-adcus2.p</td>
<td>2–40</td>
</tr>
<tr>
<td>i-adcus3.p</td>
<td>2–41</td>
</tr>
<tr>
<td>i-prmsn.p</td>
<td>2–43</td>
</tr>
<tr>
<td>i-adcus4.p</td>
<td>2–45</td>
</tr>
<tr>
<td>i-delcs2.p</td>
<td>2–45</td>
</tr>
<tr>
<td>i-itlst2.p</td>
<td>2–46</td>
</tr>
<tr>
<td>i-rept6.p</td>
<td>2–46</td>
</tr>
<tr>
<td>i-chkprm.i</td>
<td>2–47</td>
</tr>
<tr>
<td>i-secadm.p</td>
<td>2–47</td>
</tr>
<tr>
<td>i-secudp.p</td>
<td>2–48</td>
</tr>
<tr>
<td>i-action.p</td>
<td>4–20</td>
</tr>
<tr>
<td>i-intro.p</td>
<td>4–24</td>
</tr>
<tr>
<td>i-gon1.p</td>
<td>4–24</td>
</tr>
<tr>
<td>i-keys.p</td>
<td>4–25</td>
</tr>
<tr>
<td>i-gon2.p</td>
<td>4–27</td>
</tr>
<tr>
<td>i-kystka.p</td>
<td>4–29</td>
</tr>
<tr>
<td>i-kystkb.p</td>
<td>4–30</td>
</tr>
<tr>
<td>i-kystk.p</td>
<td>4–31</td>
</tr>
<tr>
<td>i-kystk2.p</td>
<td>4–32</td>
</tr>
<tr>
<td>i-kystk3.p</td>
<td>4–33</td>
</tr>
<tr>
<td>i-kystk4.p</td>
<td>4–34</td>
</tr>
<tr>
<td>i-wgetls.p</td>
<td>5–4</td>
</tr>
<tr>
<td>i-wchpr.p</td>
<td>5–5</td>
</tr>
<tr>
<td>_osprint.p</td>
<td>5–12</td>
</tr>
<tr>
<td>i-datf1.d</td>
<td>5–13</td>
</tr>
<tr>
<td>i-io3.p</td>
<td>5–14</td>
</tr>
<tr>
<td>i-dfsr.p</td>
<td>5–18</td>
</tr>
<tr>
<td>i-sstrm.p</td>
<td>5–20</td>
</tr>
<tr>
<td>File</td>
<td>Pages</td>
</tr>
<tr>
<td>--------------</td>
<td>---------------</td>
</tr>
<tr>
<td>i-dispho.p</td>
<td>5–20</td>
</tr>
<tr>
<td>i-osdir.p</td>
<td>5–26</td>
</tr>
<tr>
<td>i-osfile.p</td>
<td>5–26</td>
</tr>
<tr>
<td>i-codpag.p</td>
<td>5–28</td>
</tr>
<tr>
<td>i-export.p</td>
<td>5–30</td>
</tr>
<tr>
<td>i-datt16.d</td>
<td>5–30</td>
</tr>
<tr>
<td>i-exprt12.p</td>
<td>5–31</td>
</tr>
<tr>
<td>i-datt17.d</td>
<td>5–31</td>
</tr>
<tr>
<td>i-putdat.p</td>
<td>5–31</td>
</tr>
<tr>
<td>i-datt18.d</td>
<td>5–31</td>
</tr>
<tr>
<td>i-import.p</td>
<td>5–32</td>
</tr>
<tr>
<td>i-import2.p</td>
<td>5–32</td>
</tr>
<tr>
<td>i-datt12.d</td>
<td>5–32</td>
</tr>
<tr>
<td>i-impun1.p</td>
<td>5–33</td>
</tr>
<tr>
<td>i-datt13.d</td>
<td>5–33</td>
</tr>
<tr>
<td>i-impun2.p</td>
<td>5–33</td>
</tr>
<tr>
<td>i-datt14.d</td>
<td>5–34</td>
</tr>
<tr>
<td>i-impun3.p</td>
<td>5–34</td>
</tr>
<tr>
<td>i-clrfrnt.p</td>
<td>6–5</td>
</tr>
<tr>
<td>i-cdlial1.p</td>
<td>6–15</td>
</tr>
<tr>
<td>i-dial1.p</td>
<td>6–17</td>
</tr>
<tr>
<td>i-10-01.p</td>
<td>7–5</td>
</tr>
<tr>
<td>i-10-02.p</td>
<td>7–7</td>
</tr>
<tr>
<td>i-10-04.p</td>
<td>7–11</td>
</tr>
<tr>
<td>i-10-05.p</td>
<td>7–14</td>
</tr>
<tr>
<td>i-10-06.p</td>
<td>7–15</td>
</tr>
<tr>
<td>i-10-07.p</td>
<td>7–16</td>
</tr>
<tr>
<td>i-10-08.p</td>
<td>7–18</td>
</tr>
<tr>
<td>i-10-09.p</td>
<td>7–21</td>
</tr>
<tr>
<td>i-10-10.p</td>
<td>7–28</td>
</tr>
<tr>
<td>i-10-11.p</td>
<td>7–32</td>
</tr>
<tr>
<td>i-10-12.p</td>
<td>7–33</td>
</tr>
<tr>
<td>i-10-13.p</td>
<td>7–35</td>
</tr>
<tr>
<td>i-clpbrd.p</td>
<td>9–8</td>
</tr>
<tr>
<td>i-clpmul.p</td>
<td>9–14</td>
</tr>
<tr>
<td>r-syshlpchm.p</td>
<td>10–16</td>
</tr>
<tr>
<td>i-pipe1</td>
<td>11–7</td>
</tr>
<tr>
<td>i-pipe2</td>
<td>11–8</td>
</tr>
<tr>
<td>i-pipe2.p</td>
<td>11–8</td>
</tr>
<tr>
<td>i-do-sql.p</td>
<td>11–8</td>
</tr>
<tr>
<td>i-asksq1.c</td>
<td>11–9</td>
</tr>
<tr>
<td>i-abilip.p</td>
<td>11–16</td>
</tr>
<tr>
<td>i-cpipe.c</td>
<td>11–18</td>
</tr>
<tr>
<td>i-dillex1.p</td>
<td>12–23</td>
</tr>
<tr>
<td>i-dillex2.p</td>
<td>12–23</td>
</tr>
<tr>
<td>i-dillex3.p</td>
<td>12–24</td>
</tr>
<tr>
<td>i-ddeex2.p</td>
<td>13–9</td>
</tr>
<tr>
<td>i-ddeex1.p</td>
<td>13–18</td>
</tr>
<tr>
<td>oleauto.p</td>
<td>15–11</td>
</tr>
<tr>
<td>i-sktcv1.p</td>
<td>17–23</td>
</tr>
<tr>
<td>i-sktcl1.p</td>
<td>17–24</td>
</tr>
<tr>
<td>i-sktcv2.p</td>
<td>17–25</td>
</tr>
<tr>
<td>i-sktcl2.p</td>
<td>17–26</td>
</tr>
<tr>
<td>i-sktcv3.p</td>
<td>17–27</td>
</tr>
<tr>
<td>i-sktcl3.p</td>
<td>17–28</td>
</tr>
<tr>
<td>hlpds.c</td>
<td>18–7</td>
</tr>
<tr>
<td>hlpds.c</td>
<td>18–29</td>
</tr>
<tr>
<td>File Name</td>
<td>Page Range</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------</td>
</tr>
<tr>
<td>hlvcalc.c</td>
<td>18–30</td>
</tr>
<tr>
<td>loaddb.p</td>
<td>18–36</td>
</tr>
<tr>
<td>hlprodsp.c</td>
<td>18–37</td>
</tr>
<tr>
<td>hldemo.p</td>
<td>18–38</td>
</tr>
<tr>
<td>Calling API procedures to import audit policies in XML</td>
<td>B–5</td>
</tr>
<tr>
<td>Calling the update-auditing-policies API procedure</td>
<td>B–14</td>
</tr>
<tr>
<td>Example calls to the audit policy caching API</td>
<td>B–17</td>
</tr>
<tr>
<td>Example calls to the API for importing audit policies from XML</td>
<td>B–25</td>
</tr>
<tr>
<td>i-datf12.d</td>
<td>D–2</td>
</tr>
<tr>
<td>i-datf12.q</td>
<td>D–3</td>
</tr>
<tr>
<td>i-datf13.d</td>
<td>D–3</td>
</tr>
<tr>
<td>i-chgin3.p</td>
<td>D–4</td>
</tr>
<tr>
<td>i-datf13.q</td>
<td>D–4</td>
</tr>
<tr>
<td>i-datf14.d</td>
<td>D–5</td>
</tr>
<tr>
<td>i-chgin4.p</td>
<td>D–5</td>
</tr>
<tr>
<td>i-datf14.q</td>
<td>D–6</td>
</tr>
<tr>
<td>i-datf15.d</td>
<td>D–6</td>
</tr>
<tr>
<td>i-chgin5.p</td>
<td>D–6</td>
</tr>
</tbody>
</table>
Preface

This Preface contains the following sections:

- Purpose
- Audience
- Organization
- Using this manual
- Typographical conventions
- Examples of syntax descriptions
- Example procedures
- OpenEdge messages
- Third party acknowledgements
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 I, Data Management

Chapter 1, “Database Access”

Describes ABL statements that allow you to read and write data in a database. This chapter also describes techniques for browsing database data on a display.

Chapter 2, “Application Security”

Surveys the types of data security that ABL supports. This chapter focuses specifically on how you can provide various types of run-time security within an application.

Chapter 3, “Auditing”

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 II, Input/Output Processes

Chapter 4, “Handling User Input”

Describes how OpenEdge handles input from the keyboard and the mouse. This chapter includes examples of how you can monitor data entry from the user.

Chapter 5, “Alternate I/O Sources”

Describes how your OpenEdge application can handle I/O to and from operating system files and special devices.
Chapter 6, “Colors and Fonts”

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

Chapter 7, “Creating Reports”

Describes how to generate and format simple and complex reports.

Part III, External Program Interfaces

Chapter 8, “Introduction to External Program Interfaces”

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

Chapter 9, “System Clipboard”

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.

Chapter 10, “Providing Help for OpenEdge Applications”

Describes how to provide online help for your OpenEdge application.

Chapter 11, “Named Pipes”

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.

Chapter 12, “Shared Library and DLL Support”

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.

Chapter 13, “Windows Dynamic Data Exchange”

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.

Chapter 14, “Using COM Objects in ABL”

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

Chapter 15, “ActiveX Automation Support”

Describes OpenEdge support for ActiveX Automation and how to implement an OpenEdge application as an ActiveX Automation Controller from ABL.
Chapter 16, “ActiveX Control Support”

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.

Chapter 17, “Sockets”

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.

Chapter 18, “Host Language Call Interface”

Describes the OpenEdge Host Language Call (HLC) Interface.

Part IV, Appendices

Appendix A, “COM Object Data Type Mapping”

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

Appendix B, “Audit Policy Maintenance APIs”

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.

Appendix C, “HLC Library Function Reference”

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

Appendix D, “Command and Utility Reference”

Describes the Quoter utility.

Using this manual

OpenEdge provides a special purpose programming language for building business applications. In the documentation, the formal name for this language is ABL (Advanced Business Language). With few exceptions, all keywords of the language appear in all UPPERCASE, using a font that is appropriate to the context. All other alphabetic language content appears in mixed case.

For the latest documentation updates see the OpenEdge Product Documentation page on PSDN: http://communities.progress.com/pcom/docs/DOC-16074.
References to ABL compiler and run-time features

ABL is both a compiled and an interpreted language that executes in a run-time engine. The documentation refers to this run-time engine as the ABL Virtual Machine (AVM). When the documentation refers to ABL source code compilation, it specifies ABL or the compiler as the actor that manages compile-time features of the language. When the documentation refers to run-time behavior in an executing ABL program, it specifies the AVM as the actor that manages the specified run-time behavior in the program.

For example, these sentences refer to the ABL compiler’s allowance for parameter passing and the AVM’s possible response to that parameter passing at run time: “ABL allows you to pass a dynamic temp-table handle as a static temp-table parameter of a method. However, if at run time the passed dynamic temp-table schema does not match the schema of the static temp-table parameter, the AVM raises an error.” The following sentence refers to run-time actions that the AVM can perform using a particular ABL feature: “The ABL socket object handle allows the AVM to connect with other ABL and non-ABL sessions using TCP/IP sockets.”

References to ABL data types

ABL provides built-in data types, built-in class data types, and user-defined class data types. References to built-in data types follow these rules:

- Like most other keywords, references to specific built-in data types appear in all UPPERCASE, using a font that is appropriate to the context. No uppercase reference ever includes or implies any data type other than itself.
- Wherever integer appears, this is a reference to the INTEGER or INT64 data type.
- Wherever character appears, this is a reference to the CHARACTER, LONGCHAR, or CLOB data type.
- Wherever decimal appears, this is a reference to the DECIMAL data type.
- Wherever numeric appears, this is a reference to the INTEGER, INT64, or DECIMAL data type.

References to built-in class data types appear in mixed case with initial caps, for example, Progress.Lang.Object. References to user-defined class data types appear in mixed case, as specified for a given application example.
This manual uses the following typographical conventions:

<table>
<thead>
<tr>
<th>Convention</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Bold</strong></td>
<td>Bold typeface indicates commands or characters the user types, provides emphasis, or the names of user interface elements.</td>
</tr>
<tr>
<td><em>Italic</em></td>
<td>Italic typeface indicates the title of a document, or signifies new terms.</td>
</tr>
<tr>
<td>SMALL, BOLD CAPITAL LETTERS</td>
<td>Small, bold capital letters indicate OpenEdge key functions and generic keyboard keys; for example, <strong>GET</strong> and <strong>CTRL</strong>.</td>
</tr>
<tr>
<td>KEY1+KEY2</td>
<td>A plus sign between key names indicates a <em>simultaneous</em> key sequence: you press and hold down the first key while pressing the second key. For example, <strong>CTRL+X</strong>.</td>
</tr>
<tr>
<td>KEY1 KEY2</td>
<td>A space between key names indicates a <em>sequential</em> key sequence: you press and release the first key, then press another key. For example, <strong>ESCAPE H</strong>.</td>
</tr>
</tbody>
</table>

**Syntax:**

- **Fixed width:** A fixed-width font is used in syntax statements, code examples, system output, and filenames.
- **Fixed-width italics:** Fixed-width italics indicate variables in syntax statements.
- **Fixed-width bold:** Fixed-width bold indicates variables with special emphasis.
- **UPPERCASE fixed width:** Uppercase words are ABL keywords. Although these are always shown in uppercase, you can type them in either uppercase or lowercase in a procedure.

![This icon (three arrows) introduces a multi-step procedure.]

![This icon (one arrow) introduces a single-step procedure.]

- **Period (.) or colon (:):** All statements except **DO**, **FOR**, **FUNCTION**, **PROCEDURE**, and **REPEAT** end with a period. **DO**, **FOR**, **FUNCTION**, **PROCEDURE**, and **REPEAT** statements can end with either a period or a colon.

- **[ ]:** Large brackets indicate the items within them are optional.
- **[ ]:** Small brackets are part of ABL.
- **{ }:** Large braces indicate the items within them are required. They are used to simplify complex syntax diagrams.
- **{ }:** Small braces are part of ABL. For example, a called external procedure must use braces when referencing arguments passed by a calling procedure.
Examples of syntax descriptions

In this example, ACCUM is a keyword, and aggregate and expression are variables:

**Syntax**

```
ACCUM aggregate expression
```

FOR is one of the statements that can end with either a period or a colon, as in this example:

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

In this example, STREAM stream, UNLESS-HIDDEN, and NO-ERROR are optional:

**Syntax**

```
DISPLAY [ STREAM stream ] [ UNLESS-HIDDEN ] [ NO-ERROR ]
```

In this example, the outer (small) brackets are part of the language, and the inner (large) brackets denote an optional item:

**Syntax**

```
INITIAL [ constant [, constant ] ]
```

A called external procedure must use braces when referencing compile-time arguments passed by a calling procedure, as shown in this example:

**Syntax**

```
{ &argument-name }
```

In this example, EACH, FIRST, and LAST are optional, but you can choose only one of them:

**Syntax**

```
PRESELECT [ EACH | FIRST | LAST ] record-phrase
```
In this example, you must include two expressions, and optionally you can include more. Multiple expressions are separated by commas:

**Syntax**

```
MAXIMUM ( expression , expression [ , expression ] ... )
```

In this example, you must specify MESSAGE and at least one expression or SKIP \( [ ( n ) ] \), and any number of additional expression or SKIP \( [ ( n ) ] \) is allowed:

**Syntax**

```
MESSAGE { expression | SKIP [ ( n ) ] } ...
```

In this example, you must specify \{ include-file, then optionally any number of argument or \&argument-name = "argument-value", and then terminate with \}:

**Syntax**

```
{ include-file
  [ argument | \&argument-name = "argument-value" ] ... }
```
Long syntax descriptions split across lines

Some syntax descriptions are too long to fit on one line. When syntax descriptions are split across multiple lines, groups of optional and groups of required items are kept together in the required order.

In this example, WITH is followed by six optional items:

Syntax

WITH [ ACCUM max-length ] [ expression DOWN ]
   [ CENTERED ] [ n COLUMNS ] [ SIDE-LABELS ]
   [ STREAM-IO ]

Complex syntax descriptions with both required and optional elements

Some syntax descriptions are too complex to distinguish required and optional elements by bracketing only the optional elements. For such syntax, the descriptions include both braces (for required elements) and brackets (for optional elements).

In this example, ASSIGN requires either one or more field entries or one record. Options available with field or record are grouped with braces and brackets:

Syntax

ASSIGN  { [ FRAME frame ] { field [ = expression ] } }
       | { WHEN expression ] } ... 
       | { record [ EXCEPT field ... ] }


Example procedures

This manual provides numerous example procedures that illustrate syntax and concepts. You can access the example files and details for installing the examples from the following locations:

- Documentation and Samples directory (doc_samples) on the OpenEdge product DVD
- OpenEdge Documentation page on PSDN:

http://communities.progress.com/pcom/docs/DOC-16074

OpenEdge messages

OpenEdge displays several types of messages to inform you of routine and unusual occurrences:

- **Execution messages** inform you of errors encountered while OpenEdge is running a procedure; for example, if OpenEdge cannot find a record with a specified index field value.
- **Compile messages** inform you of errors found while OpenEdge is reading and analyzing a procedure before running it; for example, if a procedure references a table name that is not defined in the database.
- **Startup messages** inform you of unusual conditions detected while OpenEdge is getting ready to execute; for example, if you entered an invalid startup parameter.

After displaying a message, OpenEdge proceeds in one of several ways:

- Continues execution, subject to the error-processing actions that you specify or that are assumed as part of the procedure. This is the most common action taken after execution messages.
- Returns to the Procedure Editor, so you can correct an error in a procedure. This is the usual action taken after compiler messages.
- Halts processing of a procedure and returns immediately to the Procedure Editor. This does not happen often.
- Terminates the current session.

OpenEdge messages end with a message number in parentheses. In this example, the message number is 200:

```
** Unknown table name table. (200)
```

If you encounter an error that terminates OpenEdge, note the message number before restarting.
Obtaining more information about OpenEdge messages

In Windows platforms, use OpenEdge online help to obtain more information about OpenEdge messages. Many OpenEdge tools include the following Help menu options to provide information about messages:

- Choose Help→Recent Messages to display detailed descriptions of the most recent OpenEdge message and all other messages returned in the current session.

- Choose Help→Messages and then type the message number to display a description of a specific OpenEdge message.

- In the Procedure Editor, press the HELP key or F1.

On UNIX platforms, use the OpenEdge pro command to start a single-user mode character OpenEdge client session and view a brief description of a message by providing its number.

To use the pro command to obtain a message description by message number:

1. Start the Procedure Editor:

   ```
   OpenEdge-install-dir/bin/pro
   ```

2. Press F3 to access the menu bar, then choose Help→Messages.

3. Type the message number and press ENTER. Details about that message number appear.

4. Press F4 to close the message, press F3 to access the Procedure Editor menu, and choose File→Exit.
Third party acknowledgements

OpenEdge includes AdventNet - Agent Toolkit licensed from AdventNet, Inc. 
http://www.adventnet.com. All rights to such copyright material rest with AdventNet.

OpenEdge includes ANTLR (Another Tool for Language Recognition) software Copyright © 2003-2006, Terence Parr All rights reserved. Neither the name of the author nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission. Software distributed on an “AS IS” basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License agreement that accompanies the product.

OpenEdge includes software developed by the Apache Software Foundation (http://www.apache.org/). Copyright © 1999 The Apache Software Foundation. All rights reserved (Xerces C++ Parser (XML) and Xerces2 Java Parser (XML)); Copyright © 1999-2002 The Apache Software Foundation. All rights reserved (Xerces Parser (XML); and Copyright © 2000-2003 The Apache Software Foundation. All rights reserved (Ant). The names “Apache,” “Xerces,” “ANT,” and “Apache Software Foundation” must not be used to endorse or promote products derived from this software without prior written permission. Products derived from this software may not be called “Apache”, nor may “Apache” appear in their name, without prior written permission of the Apache Software Foundation. For written permission, please contact apache@apache.org. Software distributed on an “AS IS” basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License agreement that accompanies the product.

OpenEdge includes Concurrent Java software Copyright 1994-2000 Sun Microsystems, Inc. All Rights Reserved. -Neither the name of or trademarks of Sun may be used to endorse or promote products including or derived from the Java Software technology without specific prior written permission; and Redistributions of source or binary code must contain the above copyright notice, this notice and the following disclaimers: This software is provided "AS IS," without a warranty of any kind. ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT, ARE HEREBY EXCLUDED. SUN MICROSYSTEMS, INC. AND ITS LICENSORS SHALL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR DISTRIBUTING THE SOFTWARE OR ITS DERIVATIVES. IN NO EVENT WILL SUN MICROSYSTEMS, INC. OR ITS LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR DIRECT, INDIRECT, SPECIAL, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF THE USE OF OR INABILITY TO USE SOFTWARE, EVEN IF SUN MICROSYSTEMS, INC. HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

OpenEdge includes DataDirect software Copyright © 1991-2007 Progress Software Corporation and/or its subsidiaries or affiliates. All Rights Reserved. (DataDirect Connect for JDBC Type 4 driver); Copyright © 1993-2009 Progress Software Corporation and/or its subsidiaries or affiliates. All Rights Reserved. (DataDirect Connect for JDBC); Copyright © 1988-2007 Progress Software Corporation and/or its subsidiaries or affiliates. All Rights Reserved. (DataDirect Connect for ODBC); and Copyright © 1988-2007 Progress Software
Corporation and/or its subsidiaries or affiliates. All Rights Reserved. (DataDirect Connect64 for ODBC).

OpenEdge includes DataDirect Connect for ODBC and DataDirect Connect64 for ODBC software, which include ICU software 1.8 and later - Copyright © 1995-2003 International Business Machines Corporation and others All rights reserved. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation.

OpenEdge includes DataDirect Connect for ODBC and DataDirect Connect64 for ODBC software, which include software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/). Copyright © 1998-2006 The OpenSSL Project. All rights reserved. And Copyright © 1995-1998 Eric Young (eay@cryptsoft.com). All rights reserved.

OpenEdge includes DataDirect products for the Microsoft SQL Server database which contain a licensed implementation of the Microsoft TDS Protocol.

OpenEdge includes software authored by David M. Gay. Copyright © 1991, 2000, 2001 by Lucent Technologies (dtoa.c); Copyright © 1991, 1996 by Lucent Technologies (g_fmt.c); and Copyright © 1991 by Lucent Technologies (rnd_prod.s). Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which is or includes a copy or modification of this software and in all copies of the supporting documentation for such software. THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED WARRANTY. IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.

OpenEdge includes software authored by David M. Gay. Copyright © 1998-2001 by Lucent Technologies All Rights Reserved (decstrtod.c; strtodg.c); Copyright © 1998, 2000 by Lucent Technologies All Rights Reserved (decstrtof.c; strtord.c); Copyright © 1998 by Lucent Technologies All Rights Reserved (dmisc.c; gdtoa.h; gethex.c; gmisc.c; sum.c); Copyright © 1998, 1999 by Lucent Technologies All Rights Reserved (gdtoa.c; misc.c; smisc.c; ulp.c); Copyright © 1998-2000 by Lucent Technologies All Rights Reserved (gdtoaimp.h); Copyright © 2000 by Lucent Technologies All Rights Reserved (hd_init.c). Full copies of these licenses can be found in the installation directory, in the c:/OpenEdge/licenses folder. Permission to use, copy, modify, and distribute this software and its documentation for any purpose and without fee is hereby granted, provided that the above copyright notice appear in all copies and that both that the copyright notice and this permission notice and warranty disclaimer appear in supporting documentation, and that the name of Lucent or any of its entities not be used in advertising or publicity pertaining to distribution of the software without specific, written prior permission. LUCENT DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE, INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL LUCENT OR ANY OF ITS ENTITIES BE LIABLE FOR ANY SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.
OpenEdge includes http package software developed by the World Wide Web Consortium. Copyright © 1994-2002 World Wide Web Consortium, (Massachusetts Institute of Technology, European Research Consortium for Informatics and Mathematics, Keio University). All rights reserved. This work is distributed under the W3C® Software License [http://www.w3.org/Consortium/Legal/2002/copyright-software-20021231] in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

OpenEdge includes ICU software 1.8 and later - Copyright © 1995-2003 International Business Machines Corporation and others All rights reserved. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation.


OpenEdge includes Infragistics NetAdvantage for .NET v2009 Vol 2 Copyright © 1996-2009 Infragistics, Inc. All rights reserved.

OpenEdge includes JSTL software Copyright 1994-2006 Sun Microsystems, Inc. All Rights Reserved. Software distributed on an “AS IS” basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License agreement that accompanies the product.

OpenEdge includes OpenSSL software developed by the OpenSSL Project for use in the OpenSSL Toolkit (http://www.openssl.org/). Copyright © 1998-2007 The OpenSSL Project. All rights reserved. This product includes cryptographic software written by Eric Young (eay@cryptsoft.com). This product includes software written by Tim Hudson (tjh@cryptsoft.com). Copyright © 1995-1998 Eric Young (eay@cryptsoft.com) All rights reserved. The names "OpenSSL Toolkit" and "OpenSSL Project" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact openssl-core@openssl.org. Products derived from this software may not be called "OpenSSL" nor may "OpenSSL" appear in their names without prior written permission of the OpenSSL Project. Software distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License agreement that accompanies the product.

OpenEdge includes Quartz Enterprise Job Scheduler software Copyright © 2001-2003 James House. All rights reserved. Software distributed on an “AS IS” basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License agreement that accompanies the product. This product uses and includes within its distribution, software developed by the Apache Software Foundation (http://www.apache.org/).

OpenEdge includes code licensed from RSA Security, Inc. Some portions licensed from IBM are available at http://oss.software.ibm.com/icu4j/.

OpenEdge includes the RSA Data Security, Inc. MD5 Message-Digest Algorithm. Copyright ©1991-2, RSA Data Security, Inc. Created 1991. All rights reserved.
OpenEdge includes Sonic software, which includes software developed by Apache Software Foundation (http://www.apache.org/). Copyright © 1999-2000 The Apache Software Foundation. All rights reserved. The names “Ant”, “Axis”, “Xalan”, “FOP”, “The Jakarta Project”, “Tomcat”, “Xerces” and/or “Apache Software Foundation” must not be used to endorse or promote products derived from the Product without prior written permission. Any product derived from the Product may not be called “Apache”, nor may “Apache” appear in their name, without prior written permission. For written permission, please contact apache@apache.org.

OpenEdge includes Sonic software, which includes software Copyright © 1999 CERN - European Organization for Nuclear Research. Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. CERN makes no representations about the suitability of this software for any purpose. It is provided "as is" without expressed or implied warranty.

OpenEdge includes Sonic software, which includes software developed by ExoLab Project (http://www.exolab.org/). Copyright © 2000 Intalio Inc. All rights reserved. The names “Castor” and/or “ExoLab” must not be used to endorse or promote products derived from the Products without prior written permission. For written permission, please contact info@exolab.org. Exolab, Castor and Intalio are trademarks of Intalio Inc.

OpenEdge includes Sonic software, which includes software developed by IBM. Copyright © 1995-2003 International Business Machines Corporation and others. All rights reserved. Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, provided that the above copyright notice(s) and this permission notice appear in all copies of the Software and that both the above copyright notice(s) and this permission notice appear in supporting documentation. Software distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License agreement that accompanies the product. Except as contained in this notice, the name of a copyright holder shall not be used in advertising or otherwise to promote the sale, use or other dealings in this Software without prior written authorization of the copyright holder.

OpenEdge includes Sonic software, which includes the JMX Technology from Sun Microsystems, Inc. Use and Distribution is subject to the Sun Community Source License available at http://sun.com/software/communitysource.

OpenEdge includes Sonic software, which includes software developed by the ModelObjects Group (http://www.modelobjects.com). Copyright © 2000-2001 ModelObjects Group. All rights reserved. The name “ModelObjects” must not be used to endorse or promote products derived from this software without prior written permission. Products derived from this software may not be called “ModelObjects”, nor may “ModelObjects” appear in their name, without prior written permission. For written permission, please contact djacobs@modelobjects.com.

OpenEdge includes Sonic software, which includes code licensed from Mort Bay Consulting Pty. Ltd. The Jetty Package is Copyright © 1998 Mort Bay Consulting Pty. Ltd. (Australia) and others.
OpenEdge includes Sonic software, which includes files that are subject to the Netscape Public License Version 1.1 (the “License”); you may not use this file except in compliance with the License. You may obtain a copy of the License at http://www.mozilla.org/NPL/. Software distributed under the License is distributed on an “AS IS” basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License. The Original Code is Mozilla Communicator client code, released March 31, 1998. The Initial Developer of the Original Code is Netscape Communications Corporation. Portions created by Netscape are Copyright 1998-1999 Netscape Communications Corporation. All Rights Reserved.

OpenEdge includes Sonic software, which includes software developed by the University Corporation for Advanced Internet Development http://www.ucaid.edu Internet2 Project. Copyright © 2002 University Corporation for Advanced Internet Development Inc. All rights reserved. Neither the name of OpenSAML nor the names of its contributors, nor Internet2, nor the University Corporation for Advanced Internet Development, Inc., nor UCAID may be used to endorse or promote products derived from this software and products derived from this software may not be called OpenSAML, Internet2, UCAID, or the University Corporation for Advanced Internet Development, nor may OpenSAML appear in their name without prior written permission of the University Corporation for Advanced Internet Development. For written permission, please contact opensaml@opensaml.org.

OpenEdge includes the UnixWare platform of Perl Runtime authored by Kiem-Phong Vo and David Korn. Copyright © 1991, 1996 by AT&T Labs. Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which is or includes a copy or modification of this software and in all copies of the supporting documentation for such software. THIS SOFTWARE IS BEING PROVIDED “AS IS”, WITHOUT ANY EXPRESS OR IMPLIED WARRANTY. IN PARTICULAR, NEITHER THE AUTHORS NOR AT&T LABS MAKE ANY REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.

OpenEdge includes Vermont Views Terminal Handling Package software developed by Vermont Creative Software. Copyright © 1988-1991 by Vermont Creative Software.

OpenEdge includes XML Tools, which includes versions 8.9 of the Saxon XSLT and XQuery Processor from Saxonica Limited (http://www.saxonica.com/) which are available from SourceForge (http://sourceforge.net/projects/saxon/). The Original Code of Saxon comprises all those components which are not explicitly attributed to other parties. The Initial Developer of the Original Code is Michael Kay. Until February 2001 Michael Kay was an employee of International Computers Limited (now part of Fujitsu Limited), and original code developed during that time was released under this license by permission from International Computers Limited. From February 2001 until February 2004 Michael Kay was an employee of Software AG, and code developed during that time was released under this license by permission from Software AG, acting as a "Contributor". Subsequent code has been developed by Saxonica Limited, of which Michael Kay is a Director, again acting as a "Contributor". A small number of modules, or enhancements to modules, have been developed by other individuals (either written especially for Saxon, or incorporated into Saxon having initially been released as part of another open source product). Such contributions are acknowledged individually in comments attached to the relevant code modules. All Rights Reserved. The contents of the Saxon files are subject to the Mozilla Public License Version 1.0 (the "License"); you may not use these files except in compliance with the License. You may obtain a copy of the License at http://www.mozilla.org/MPL/ and a copy of the license can also be found in the
installation directory, in the c:/OpenEdge/licenses folder. Software distributed under the License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License.

OpenEdge includes XML Tools, which includes Xs3P v1.1.3. The contents of this file are subject to the DSTC Public License (DPL) Version 1.1 (the "License"); you may not use this file except in compliance with the License. A copy of the license can be found in the installation directory, in the c:/OpenEdge/licenses folder. Software distributed under the License is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for the specific language governing rights and limitations under the License. The Original Code is xs3p. The Initial Developer of the Original Code is DSTC. Portions created by DSTC are Copyright © 2001, 2002 DSTC Pty Ltd. All rights reserved.

OpenEdge includes YAJL software Copyright 2007, Lloyd Hilaiel. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met: 1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer. 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution. 3. Neither the name of Lloyd Hilaiel nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission. THIS SOFTWARE IS PROVIDED BY THE AUTHOR ``AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
Part I

Data Management

Chapter 1, Database Access
Chapter 2, Application Security
Chapter 3, Auditing
This chapter describes how to access records in an OpenEdge® database. This chapter covers 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 ORACLE. You must connect to a database before you can access it. There are four ways to connect to a database:

- As an argument when starting OpenEdge
- With the CONNECT statement (in the Procedure Editor or in an ABL—Advanced Business Language—procedure)
- With the OpenEdge Data Dictionary
- 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 the “Connection failures and disruptions” section on page 1–5. For more information on database connection management, see OpenEdge Deployment: Startup Command and Parameter Reference and OpenEdge Deployment: Managing ABL Applications.

Connection parameters

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

You can also create secure database connections using the SSL-based Connections (-ssl) startup parameter. For more information, see Chapter 2, “Application Security.”
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 *OpenEdge Deployment: 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 *OpenEdge Deployment: 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
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 *OpenEdge Development: 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 *OpenEdge Development: ABL Reference*.

For information on how to set up an auto-connect list in the primary database, see *OpenEdge Deployment: Managing 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 the “Logical database names” section on page 1–10.

Run-time connection considerations

As mentioned in the “Database connections” section on page 1–2, 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).

Table 1–1 lists the default failure behavior for each of the run-time connection techniques.

Table 1–1: Connection failure behavior

<table>
<thead>
<tr>
<th>Connection technique</th>
<th>Default connection failure behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>At OpenEdge startup</td>
<td>The OpenEdge session does not run.</td>
</tr>
<tr>
<td>CONNECT statement</td>
<td>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.</td>
</tr>
<tr>
<td>Auto-connect</td>
<td>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.</td>
</tr>
</tbody>
</table>
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. Although the NO–ERROR option bypasses ordinary error processing, OpenEdge still displays an error message. For more information on using the NO–ERROR option, see *OpenEdge Development: 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, you 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:

```abl
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 *OpenEdge Development: ABL Reference*. 
**OpenEdge database connection functions**

Table 1–2 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 1–2: ABL database functions**

<table>
<thead>
<tr>
<th>ABL function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONNECTED</td>
<td>Tests whether a given database is connected</td>
</tr>
<tr>
<td>DATASERVERS</td>
<td>Returns a character string containing a list of database types supported by</td>
</tr>
<tr>
<td></td>
<td>the installed OpenEdge product; for example, “OpenEdge,ORACLE.”</td>
</tr>
<tr>
<td>DB–REMOTE–HOST</td>
<td>Returns a character string containing the IP address of the database</td>
</tr>
<tr>
<td></td>
<td>connection</td>
</tr>
<tr>
<td>DBRESTRICTIONS</td>
<td>Returns a character string that describes the OpenEdge features that are</td>
</tr>
<tr>
<td></td>
<td>not supported for a particular database, for example, if the database is an</td>
</tr>
<tr>
<td></td>
<td>ORACLE database, the return string is: “LAST,PREV,RECID,SETUSERID”</td>
</tr>
<tr>
<td>DBTYPE</td>
<td>Returns the database type of a currently connected database, for example</td>
</tr>
<tr>
<td></td>
<td>“OpenEdge,” “ORACLE,” etc.</td>
</tr>
<tr>
<td>DBVERSION</td>
<td>Returns the version of the currently connected database</td>
</tr>
<tr>
<td>FRAME–DB</td>
<td>Returns a character string that contains the logical name of the database</td>
</tr>
<tr>
<td></td>
<td>for the field in which the cursor was last positioned for input</td>
</tr>
<tr>
<td>NUM–DBS</td>
<td>Returns the number of connected databases</td>
</tr>
<tr>
<td>LDBNAME</td>
<td>Returns the logical name of a currently connected database</td>
</tr>
<tr>
<td>PDBNAME</td>
<td>Returns the physical name of a currently connected database</td>
</tr>
<tr>
<td>SDBNAME</td>
<td>Returns the logical name of a schema holder for a database</td>
</tr>
</tbody>
</table>

For more information on these functions, see *OpenEdge Development: 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 FIELDS(Name Address City State PostalCode) EXCLUSIVE-LOCK:
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 *OpenEdge Deployment: 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**

```plaintext
/* 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**

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

**i-dispcust.p**

```plaintext
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**

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

**i-makebuf.p**

```plaintext
DEFINE NEW SHARED BUFFER mybuf FOR myalias.customer.
CREATE ALIAS myalias FOR DATABASE sports2.
RUN i-disp.p
```

**i-disp.p**

```plaintext
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.

*Figure 1–1* shows how the data-handling statements move data.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Database record</th>
<th>Record buffer</th>
<th>Screen buffer</th>
<th>User</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASSIGN</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CREATE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DELETE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DISPLAY</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENABLE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FIND</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FOR EACH</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GET</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INSERT</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OPEN QUERY (with BROWSE)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PROMPT-FOR</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RELEASE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>REPOSITION (with BROWSE)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SET</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>UPDATE</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Figure 1–1:* Data movement

To use *Figure 1–1*, 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).

Figure 1–2 shows the primary data-handling statements and shows which statements are composed of other statements.

**Figure 1–2: 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 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 Figure 1–2, see *OpenEdge Development: 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 OpenEdge Development: 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):

```
```

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. Table 1–3 summarizes the differences in record fetching between the FIND, FOR EACH, OPEN QUERY, and PRESELECT statements.

Table 1–3: Record fetching

<table>
<thead>
<tr>
<th>Statement to define set of records</th>
<th>Statement to fetch the records</th>
<th>Structure used to locate records</th>
</tr>
</thead>
<tbody>
<tr>
<td>FIND</td>
<td>FIND</td>
<td>Index</td>
</tr>
<tr>
<td>FOR EACH</td>
<td>FOR EACH</td>
<td>Results List or Index</td>
</tr>
<tr>
<td>OPEN QUERY</td>
<td>GET</td>
<td>Results List or Index</td>
</tr>
<tr>
<td>DO PRESELECT or REPEAT PRESELECT</td>
<td>FIND</td>
<td>Results List</td>
</tr>
</tbody>
</table>

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 *OpenEdge Getting Started: ABL Essentials*. 
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:

```abl
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:

```abl
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:

```abl
DEFINE TEMP-TABLE ttRid NO-UNDO
    FIELD ridfld AS CHARACTER.
FOR EACH Customer FIELDS (balance) 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 the “Results lists” section on page 1–23.

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 1–3, 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.

Table 1–4 summarizes how the results list is built for each type of query.

Table 1–4: Results lists for specific query types

<table>
<thead>
<tr>
<th>Query type</th>
<th>Results list</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-scrolling, index-sorted, no preselection</td>
<td>None</td>
</tr>
<tr>
<td>Scrolling, no sorting, no preselection</td>
<td>Empty list(^1) established when query is opened—records are added to the results list as needed</td>
</tr>
<tr>
<td>Presorted or preselected</td>
<td>Complete list built when query is opened</td>
</tr>
</tbody>
</table>

\(^1\) If a browse is defined for the query, the results list initially contains one row.
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.

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 Table 1–3, 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 */
GET PREV q. /* query is positioned ON cust 2 */
REPOSITION q FORWARD 0. /* query is positioned BETWEEN cust 1 and 2 */
GET NEXT q. /* gets cust no. 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 Table 1–3 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:
END.
```
Within a PRESELECT block, you can use the FIND statement to move backwards and forwards through the results list:

```plaintext
/* 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.
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:

```plaintext
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:

```plaintext
FIND FIRST Customer NO-LOCK.
PAUSE.
FIND NEXT Customer NO-LOCK USE-INDEX country-post.
```

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.
Pause.
FIND NEXT Customer NO-LOCK WHERE Customer.CustNum > 1.
```

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 cust–num 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 *OpenEdge Data Management: SQL Development.*

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
  ELSE LEAVE.
  END.
END.
```

`i-fldls2.p`

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

```plaintext
FOR EACH Customer FIELDS (Name),
  EACH Invoice FIELDS (InvoiceNum Amount) OF Customer:
```

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 OpenEdge Deployment: Managing 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 OpenEdge Deployment: Managing 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:

```alb
FOR EACH Customer FIELDS (Name 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 the “**FIND repositioning**” section on page 1–27.) 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 *OpenEdge Getting Started: ABL Essentials*.

- **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".

Figure 1–3 shows how you can join three tables.

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 Figure 1–3 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. Figure 1–3 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. Figure 1–4 shows an example of left outer joins using the same tables as in Figure 1–3.

**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 the “Mixing inner and left outer joins” section on page 1–40.

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 *OpenEdge Getting Started: ABL Essentials*. 

---

Joining tables
The following code fragment generates the left outer joins shown in Figure 1–4:

```sql
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 Figure 1–4 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.
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

```abl
DEFINE QUERY q1 FOR Customer FIELDS(Name CreditLimit),
    Order FIELDS(OrderNum), OrderLine FIELDS(Price Qty),
    Item FIELDS(ItemName).
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 Figure 1–5. 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 1–5: Inner join example](image)

**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 the “Using inner joins” section on page 1–37) except that all the joins are left outer joins instead of inner joins.

**i-join2.p**

```plaintext
DEFINE QUERY q1 FOR Customer FIELDS(Name CreditLimit),
              Order FIELDS(OrderNum), OrderLine FIELDS(Price Qty),
              Item FIELDS(ItemName).
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 Figure 1–6. In this example, you see the same golf club order as in Figure 1–5 along with many other orders that do not meet the selection criteria and some Customers who have no orders at all.

![Figure 1–6: Left outer join example](image)

**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 OpenEdge Getting Started: ABL Essentials.
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.
The CONTAINS operator

You can use the CONTAINS operator of the WHERE option of the record phrase to query an 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 OpenEdge Development: Internationalizing 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 OpenEdge Development: Basic Development Tools or 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 OpenEdge Deployment: 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 OpenEdge Development: Internationalizing 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 Table 1–5.

Table 1–5:  **Is the dot a word delimiter?**

<table>
<thead>
<tr>
<th>Character string</th>
<th>Function of the dot</th>
<th>Is the dot a word delimiter?</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Balance is $25,125.95”</td>
<td>Decimal separator</td>
<td>No</td>
</tr>
<tr>
<td>“Shipment not received. Call customs broker”</td>
<td>Period at end of sentence</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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 Table 1–6.

Table 1–6:  **Word delimiter attributes**  

<table>
<thead>
<tr>
<th>Word delimiter attribute</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>LETTER</td>
<td>Always part of a word</td>
<td>Assigned to all characters that the current attribute table defines as letters.</td>
</tr>
<tr>
<td></td>
<td>In English, these are the uppercase characters A–Z and the lowercase characters a–z.</td>
<td></td>
</tr>
<tr>
<td>DIGIT</td>
<td>Always part of a word</td>
<td>Assigned to the characters 0–9.</td>
</tr>
<tr>
<td>USE_IT</td>
<td>Always part of a word</td>
<td>Assigned to the following characters:</td>
</tr>
<tr>
<td></td>
<td>• Dollar sign ($)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Percent sign (%)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Number sign (#)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• At symbol (@)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Underline (_)</td>
<td></td>
</tr>
<tr>
<td>BEFORE_LETTER</td>
<td>Part of a word only if followed by a character with the LETTER attribute; otherwise, treated as a word delimiter</td>
<td>–</td>
</tr>
</tbody>
</table>
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:

```c
/* 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,
};
/* this is a comment */
```

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/pro1ang/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:

```
<table>
<thead>
<tr>
<th>Operating system</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIX</td>
<td>proutil -C wbreak-compiler src-file rule-num</td>
</tr>
<tr>
<td>Windows</td>
<td></td>
</tr>
</tbody>
</table>
```

*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:

```
<table>
<thead>
<tr>
<th>Operating system</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIX</td>
<td>proutil database -C word-rules rule-num</td>
</tr>
<tr>
<td>Windows</td>
<td></td>
</tr>
</tbody>
</table>
```

*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 *OpenEdge Development: Internationalizing 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 syntax of PROUTIL with the IDXBUILD qualifier is:

<table>
<thead>
<tr>
<th>Operating system</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIX Windows</td>
<td>proutil db-name -C idxbuild [ all ] [ -T dir-name ] [ -TB blocksize ] [ -TM n ] [ -B n ]</td>
</tr>
</tbody>
</table>

The syntax of PROUTIL with the IDXFIX qualifier is:

<table>
<thead>
<tr>
<th>Operating system</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNIX Windows</td>
<td>proutil db-name -C idxfix</td>
</tr>
</tbody>
</table>

For more information on the PROUTIL utility, see OpenEdge Data Management: Database
Administration.
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 PROWDrule-num. For example, if the compiled word-break table has the name provord.34 and resides in the DLC/mydir/mysubdir directory, set the environment variable PROW34 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:

Syntax

```
```

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 Customer.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:
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:

```abl
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
- Creating and maintaining sequences
- Accessing and incrementing sequences
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. Figure 1–7 shows the basic relationship between database fields and sequences.

![Database fields and sequences comparison](image)

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:

```assembly
... 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 EXCEPT Customer.CustNum.
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.

Table 1–7 compares sequences and control tables.

<table>
<thead>
<tr>
<th>Capability</th>
<th>Sequences</th>
<th>Control tables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Access speed</td>
<td>Fast</td>
<td>Slow</td>
</tr>
<tr>
<td>Transaction independent</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Guaranteed order</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Auto initializing</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Auto cycling</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Bounds checking</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Database limit</td>
<td>250 per 1K database block size</td>
<td>Field limit</td>
</tr>
</tbody>
</table>
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.

Figure 1–8 shows how two overlapping transactions with rollback can create records and commit consecutive sequence values with gaps.

![Figure 1–8: Sequences and overlapping transactions](image)
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 the “Using the CURRENT–VALUE statement” section on page 1–61.

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 *OpenEdge Development: Basic Development Tools* (Character only) and, in graphical interfaces, the online Help for the OpenEdge Data Dictionary.

For information on using sequences in ABL, see the “Accessing and incrementing sequences” section on page 1–58.

**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 *OpenEdge Data Management: Database Administration.*
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 OpenEdge Data Management: Database Administration.

Accessing and incrementing sequences

Figure 1–8 lists ABL statements and functions you can use to access and increment sequence values.

Table 1–8: Sequence statements and functions

<table>
<thead>
<tr>
<th>Statement or function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURRENT–VALUE function</td>
<td>Returns the current value of a sequence</td>
</tr>
<tr>
<td>DYNAMIC–CURRENT–VALUE function</td>
<td>Returns the current value of a dynamically-specified sequence</td>
</tr>
<tr>
<td>NEXT–VALUE function</td>
<td>Increments and returns the incremented value for a sequence</td>
</tr>
<tr>
<td>DYNAMIC–NEXT–VALUE function</td>
<td>Increments and returns the incremented value for a dynamically-specified sequence</td>
</tr>
<tr>
<td>CURRENT–VALUE statement</td>
<td>Sets a new current value for a sequence</td>
</tr>
<tr>
<td>DYNAMIC–CURRENT–VALUE statement</td>
<td>Sets a new current value for a dynamically-specified sequence</td>
</tr>
</tbody>
</table>

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 sequence statements and functions have the following syntax:

**Syntax**

```
CURRENT–VALUE
   (sequence [ , logical-dbname ] ) [ = expression ]
```

```
NEXT–VALUE
   (sequence [ , logical-dbname ] )
```

sequence

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.

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

Syntax

DYNAMIC-CURRENT-VALUE

\[ (\text{sequence-expression}, \text{logical-dbname-expression}) [= \text{expression}] \]

Syntax

DYNAMIC-NEXT-VALUE

\[ (\text{sequence-expression}, \text{logical-dbname-expression}) \]

sequence-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.

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 OpenEdge Development: 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:

```plaintext
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:

```plaintext
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.
Customer.CustNum = input-custnum.
...
```

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:

```openedge
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 *OpenEdge Data Management: Database Administration*.
Compiling procedures that reference sequences

When compiling procedures that use sequence statements and functions, do **not** run OpenEdge with the Time Stamp (-tstamp) startup parameter. All references to CURRENT–VALUE and NEXT–VALUE statements and functions in the r-code depend on CRC calculations made with the _Sequence metaschema table, and are not available with time stamping. CRC validation is now the default data consistency option for OpenEdge.

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 an 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:

```ABL
/* 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:

```ABL
/* 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:

```plaintext
/* 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入り`):

```
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.

```
IF NOT CONNECTED("db1")
THEN DO:
  MESSAGE "CONNECTING db1".
  CONNECT db1 -1 NO-ERROR.
END.
IF CONNECTED("db1")
THEN RUN db1proc.p.
ELSE MESSAGE "db1proc.p NOT AVAILABLE".
IF NOT CONNECTED("db2")
THEN DO:
  MESSAGE "CONNECTING db2".
  CONNECT db2 -1 NO-ERROR.
END.
IF CONNECTED("db2")
THEN RUN db2proc.p.
ELSE MESSAGE "db2proc.p NOT AVAILABLE".
```

Figure 1–9: Positioning database references

Figure 1–9 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 OpenEdge Development: 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**

```plaintext
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.
   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

```plaintext
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 NO-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 *OpenEdge Deployment: Managing ABL Applications*, and for a description of `–cache`, see *OpenEdge Deployment: Startup Command and Parameter Reference.*
As a developer, you can provide your OpenEdge® applications with application security that prevents unauthorized users from running application procedures or accessing data. The following sections describe how you can manage application security features using Advanced Business Language (ABL):

- Types of application security
- Authenticating user identities
- Managing identities
- Using secure database connections
- Authorizing access to procedures and database resources
- Using cryptography to secure data
Types of application security

OpenEdge provides the following types of application security:

- **Authentication and identity management** provides support for user authentication and ensures that an authenticated user ID is valid for authorization purposes.

- **Connection authorization** ensures that only authorized clients can connect to the database and that the communication between the client and the database is secure from unauthorized interception.

- **Schema authorization** ensures that only authorized users can modify table, field, and index definitions.

- **Compile-time authorization** ensures that only authorized users can compile procedures that access specific database tables and fields.

- **Run-time authorization** ensures that only authorized users can access specific database tables and fields or run specific precompiled procedures.

- **Cryptography** ensures data privacy and integrity so that unauthorized users cannot read or change data.

The security administrator can define security for connection, schema, and compile-time authorization in the Data Dictionary and in other tools to manage secure database connections. OpenEdge also uses the same settings for compile-time authorization to handle run-time authorization to access database tables and fields. For more information on database security administration, see *OpenEdge Data Management: Database Administration* and *OpenEdge Deployment: Managing ABL Applications*.

As described in the rest of this chapter, the developer is usually responsible for authenticating users, managing user identity, connecting securely to a database, authorizing application resources and rights to access an account, and managing cryptographically secured data.
Authenticating user identities

*Authentication* is the process of verifying a user’s identity. OpenEdge provides support for two basic mechanisms to authenticate user identities (*user IDs*):

- The OpenEdge internal authentication system (*_User* table)
- External authentication systems

The OpenEdge *internal authentication system* uses the *_User* table provided by every OpenEdge RDBMS to store user IDs and passwords. OpenEdge provides ABL elements and options to authenticate input user IDs against the values stored in this table.

The result of authenticating a user ID using OpenEdge internal authentication automatically assigns a database connection ID for the connected database. Thus, the OpenEdge internal authentication system provides automatic user identity management for a given database. For more information on database connection IDs, see the “Managing identities” section on page 2–7.

An *external authentication system* can consist of any mechanism that you choose to authenticate user IDs and that you can implement from within ABL. It can be as simple as a user table that you control and maintain or as complex as an industry-standard authentication system, such as Lightweight Directory Access Protocol (LDAP), to which you provide an ABL-based interface.

The result of authenticating a user ID using an external authentication system depends entirely on how your application supports identity management. First and foremost, you can use an externally authenticated user ID as an OpenEdge session ID, independent of database connections, to authorize access to your own application features. However, you can also use an externally authenticated user ID to set a database connection ID. For more information on user identities and for some examples of using external authentication systems, see the “Managing identities” section on page 2–7.

The following sections describe the basic requirements for authenticating user IDs and making use of the in ABL applications:

- **Specifying OpenEdge internal user IDs and passwords**
- **Authenticating OpenEdge internal user IDs and passwords**
- **Using externally-authenticated user IDs and passwords**

**Specifying OpenEdge internal user IDs and passwords**

OpenEdge internal user IDs and passwords define user identities for the OpenEdge internal authentication system, and they are used to specify a database connection ID. You must specify them in the *_User* table of each OpenEdge RDBMS using the user maintenance options of the *Admin→Security* menu in the Data Administration and character-mode Data Dictionary tools (see the Data Administration online help). The following sections describe the conventions for specifying OpenEdge internal user IDs and passwords.
OpenEdge internal user IDs

An OpenEdge internal user ID is a string of up to 32 characters associated with a particular OpenEdge database connection. Like table names, OpenEdge internal user IDs must begin with a character from a–z or from A–Z. The name can consist of:

- Alphabetic characters
- Digits
- The following special characters: # $ % & – _

OpenEdge internal user IDs are not case sensitive: they can be uppercase, lowercase, or any combination of these. You can establish the valid user IDs for a database through the OpenEdge Data Dictionary.

An OpenEdge internal user ID can be blank, written as the string, " ", but it cannot be defined as such through the Data Dictionary.

Passwords

An OpenEdge internal password is a string of up to 16 characters that is associated with a user ID. When you add a password through the Data Dictionary, the password is encoded with the ENCODE function. All OpenEdge internal passwords are case sensitive. So, ENCODE generates different values for uppercase and lowercase input.

When using ENCODE with passwords, Progress Software Corporation strongly recommends:

- Occurrences of the ENCODE function related to the same password run in the same code page.
- In environments with multiple code pages, programs use the CODEPAGE–CONVERT function so that occurrences of the ENCODE function related to the same password run in the same code page.

Authenticating OpenEdge internal user IDs and passwords

If the security administrator establishes a list of valid OpenEdge internal user IDs, then your application must prompt the user for a user ID and password at the appropriate point to establish a database connection ID. Typically, an application does this by running the standard OpenEdge startup procedure, _prostar.p. This procedure, in turn, runs the standard ABL authentication procedure, _login.p, for each connected database.

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.i }
DEFINE INPUT PARAMETER viewAsDialog AS LOGICAL NO-UNDO.
DEFINE VARIABLE tries AS INTEGER NO-UNDO.
IF USERID("DICTDB") <> "" OR NOT CAN-FIND(FIRST DICTDB._User) THEN RETURN.
DO ON ENDKEY UNDO, LEAVE:
ASSIGN
  currentdb = LDBNAME("DICTDB")
  // reset id and password to blank in case of retry */
  id = ""
  password = "".
IF viewAsDialog THEN DO:
  DISPLAY currentdb WITH FRAME logindb_frame VIEW-AS DIALOG-BOX.
  UPDATE id password ok_btn cancel_btn help_btn
    WITH FRAME logindb_frame view-as dialog-box.
END.
ELSE DO:
  DISPLAY currentdb WITH FRAME login_frame.
  UPDATE id password ok_btn cancel_btn help_btn
    WITH FRAME login_frame.
END.
IF SETUSERID(id,password,"DICTDB") <> TRUE 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.
HIDE FRAME login_frame.
```

The _login.p procedure uses the ABL SETUSERID function to check the user ID and password that the user enters. 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 database. If the user ID and password combination is valid for the database, SETUSERID establishes that user ID as the database connection ID.

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 earlier, 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 for the database name.

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), then the user is assigned the blank user ID. While blank user IDs can connect to the database, they cannot access data protected by compile-time and run-time authorization.
You can configure how the database handles blank user IDs for both database connection and authorization for data access. The Security menu of the Data Administration tool provides an option to disallow blank user ID access to data, which procedure access at both compile-time and run-time. This menu also provides set of additional database options, including an option to disallow database connections using the blank user ID. For more information, see the sections on security in OpenEdge Deployment: Managing ABL Applications and the Data Administration tool online help.

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 SETUSERID function directly after the connection is already established.

The i-passts.p procedure connects to the mywork database that has a list of valid users. The user initially connects to the database with a blank user ID. The code then enters a loop that forces the user to provide a valid user ID and password for that database.

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".

CONNECT mywork.

DO WHILE NOT success:
    MESSAGE "Enter a user ID and password for the database mywork.".
    SET user-id passwd BLANK.
    IF SETUSERID(user-id, passwd, "mywork") THEN
        success = TRUE.
    ELSE DO:
        BELL.
        MESSAGE "Invalid user ID and password; please try again.".
    END.
END.
```

Using externally-authenticated user IDs and passwords

The requirements for user IDs and passwords vary according to the authentication system that defines them. The only requirement that affects an OpenEdge application is where you configure user IDs for database and other OpenEdge authorization purposes. User IDs defined for OpenEdge authorization features are limited in length to the maximum length of the ABL CHARACTER data type. For more information, see the data types section in OpenEdge Development: ABL Reference.
Managing identities

OpenEdge supports different types of user identities for different purposes. Depending on how you authenticate and set a user ID, and the database options you configure, that user ID can assume one of four possible identities:

- **Database connection identity** — A database connection ID is a user ID that is always associated with a specific connected OpenEdge RDBMS. An OpenEdge RDBMS authorizes all database connections and access specific database tables and fields using the database connection ID. You can set the database connection ID from a user ID that is authenticated using either the OpenEdge internal authentication system (User table) or using an external authentication system. The ABL USERID function returns the current database connection ID for a database connection, regardless of how it is set.

- **OpenEdge session identity** — An OpenEdge session ID is a user ID that is associated with a given ABL session, independent of any database connections. The OpenEdge session ID can thus be used to authorize or identify 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 next bullet). You can set the OpenEdge session ID from a user ID that is authenticated using an external authentication system.

- **Application user identity** — An application user ID is a common user ID established by an n-tier application for use by all OpenEdge sessions that participate in handling a single user action or request. Typically, the application user ID is shared between a single OpenEdge AppServer client and the AppServer agent or agents that process client requests. Depending on the application session model, this single application user ID can also be shared between a single OpenEdge client session and multiple AppServer instances. Any given OpenEdge session can use the application user ID to set the OpenEdge session ID and any or all database connection IDs required by that session. You can set the application user ID from a user ID that is provided by a single controlling (typically client) session and that is authenticated using an external authentication system. For more information on n-tier applications, the OpenEdge AppServer, and application session models, see OpenEdge Getting Started: Application and Integration Services.

- **Auditing identity** — An auditing ID is the designated user ID that OpenEdge auditing records in audit event records for an audit trail. There is no functionally independent auditing ID. Instead, the auditing ID is set from one of the other established identities, depending on application configuration. By default, the auditing ID for the audit trail recorded by a given database is the database connection ID for that database. However, you can also set a database option to set the auditing ID from the OpenEdge session ID of any OpenEdge session that connects to the database. In this way, you can configure auditing for every database that is connected from a given OpenEdge session so that all audit trails for that session are associated with the same user ID. For more information on auditing identity, see OpenEdge Getting Started: Core Business Services.

In any given application, you might use none, some, or all of these user identities to secure or 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 n-tier. ABL provides tools to handle both.

ABL also provides functions for creating and managing unique identities for software objects. OpenEdge creates its own unique identities, for example, for audit data and physical databases.
You can also use these functions to generate unique identities for more abstract objects that you maintain, such as client sessions and other objects associated with your application.

The following sections describe how to establish and manage the identities supported in OpenEdge:

- ABL for managing user identities
- Client-principal objects
- Managing client login sessions
- Managing database connection IDs
- Managing OpenEdge session IDs
- Managing application user IDs in n-tier applications
- Managing auditing IDs
- Creating and managing unique object identities

**ABL for managing user identities**

ABL for managing user identities consists of the elements shown in Table 2–1.

**Table 2–1: ABL elements for managing user identities**

<table>
<thead>
<tr>
<th>This ABL element . . .</th>
<th>Provides this identity management function . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>SETUSERID function</td>
<td>Authenticates and sets a specified user ID as the database connection ID using the _User table of a connected OpenEdge RDBMS.</td>
</tr>
<tr>
<td>Client-principal object handle</td>
<td>Starts and maintains a client login session for a previously authenticated user ID. The client principal object contains various information about the login session, including information necessary to validate and set the user identity according to an appropriate trusted domain registry.¹²</td>
</tr>
<tr>
<td>CREATE CLIENT-PRINCIPAL statement</td>
<td>Creates a client-principal object used to start and maintain a given client login session.</td>
</tr>
<tr>
<td>SET-DB-CLIENT function</td>
<td>Given a user ID provided with a specified client-principal object and a connected OpenEdge RDBMS, asserts and validates the user ID as a database connection ID for the connected database using a database trusted domain registry.¹</td>
</tr>
</tbody>
</table>
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 trusted domain registries and how they are used in OpenEdge, see the section on identity management in *OpenEdge Getting Started: Core Business Services*.

### Client-principal objects

If you use the OpenEdge internal authentication system (see the “Authenticating OpenEdge internal user IDs and passwords” section on page 2–4) to authenticate and set a database connection ID, the specified database authorizes access according to this user ID. Also, by default, with auditing enabled, the database records audit events associated with this user ID. However, no other information about the user session is maintained for later review.

If you use your own external authentication system to authenticate user IDs for your application, you might still have to authenticate a database connection ID (the same or different from the OpenEdge session ID) in order to obtain authorization to connect to a given database. The database connection ID, whatever it is, continues to be the auditing ID, for any audit events that your application generates. And, there is no information about the application user session maintained for later review, except what is associated with the database connection ID.
A client-principal object allows you to establish a client login session for a given user ID, no matter how you authenticate that ID. A client 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. These states are controlled by methods and other attribute settings of the client-principal object handle. Such states include whether the session is logged in or out, or has been suspended, expired, or otherwise failed. Thus, the client-principal object, provides the following application features:

- Maintains a run-time client 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 client login session, such as the user 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 assert and validate a user ID against a trusted domain registry, and there by set a particular type of identity from that user ID according to the type of domain registry (database or application) and associated database option settings
- Can be exported and imported to transport a 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 ID once and set that same ID many times, in many different places, thus achieving the goal of single sign-on for your entire application.

**Client-principal object attributes**

The client-principal object provides several attributes that you can set for various functional and informational purposes. For example, you can set an automatic expiration time for the client-principal object (and the client login session it controls). You cannot use the resulting client-principal object to assert a user identity until you seal the object using the SEAL( ) method (see Table 2–4). Whether you set them all or not, the client-principal object attributes become read-only once you have sealed the object.
For a given client-principal object, OpenEdge sets the read-only values of several attributes on behalf of the application, according to client-principal object method calls and other actions. Table 2–2 describes these attributes, listed in general usage order. For more information on how these attributes are set, see Table 2–3 and Table 2–4.

**Table 2–2: Client-principal object attributes you can read-only**

<table>
<thead>
<tr>
<th>This attribute . . .</th>
<th>Returns this value . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOGIN-STATE</td>
<td>A CHARACTER value that represents the state of the client-principal object and any client login session started with it. The default value is &quot;LOGOUT&quot;.</td>
</tr>
<tr>
<td>STATE-DETAIL</td>
<td>A CHARACTER value that provides more detailed information about the most recent setting of the LOGIN-STATE attribute. ABL also sets this to any reason parameter value that you pass to the AUTHENTICATION-FAILED( ) method.</td>
</tr>
<tr>
<td>SEAL-TIMESTAMP</td>
<td>A DATETIME-TZ value that represents the time when the client-principal object was sealed (client session was logged in). ABL sets this value when you call the SEAL( ) method.</td>
</tr>
</tbody>
</table>

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 requirements, as described and listed in Table 2–3 in general usage order. These settings are all application defined. Unless otherwise noted, these attributes hold CHARACTER values. All references to a “user” indicate the user associated with the user ID that this client-principal object represents.

**Table 2–3: Client-principal object attributes you can set (1 of 3)**

<table>
<thead>
<tr>
<th>This attribute . . .</th>
<th>Sets this value . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>Required attributes</td>
<td></td>
</tr>
<tr>
<td>SESSION-ID</td>
<td>(Required) A value to uniquely identify the client login session for the user represented by the client-principal object. You must set this value before sealing the object (logging in the client session). You can set this to any value, but you might want to ensure that the value is unique in case you use it as a reference to distinguish it among multiple client login sessions. To obtain a unique value, you can use the ABL GENERATE-UUID function (see the “Creating and managing unique object identities” section on page 2–36). If you want to associate the login session with a local AppServer session, you can also use the SERVER-CONNECTION-ID attribute on the SESSION system handle to obtain a value. For more information on when you might do this, see the “Managing application user IDs in n-tier applications” section on page 2–28.</td>
</tr>
</tbody>
</table>
### Table 2–3: Client-principal object attributes you can set

<table>
<thead>
<tr>
<th>This attribute . . .</th>
<th>Sets this value . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>USER-ID</strong></td>
<td>(Required) The authenticated user ID of the user represented by this client-principal object. You must set this value before sealing the object (logging in the client session). <strong>Note:</strong> For validation purposes, it is up to you to ensure that the <strong>DOMAIN-NAME</strong> attribute is also set to the name of the authentication system (domain) that you actually used to authenticate this user ID.</td>
</tr>
</tbody>
</table>
| **DOMAIN-NAME**      | (Required) The name of the authentication system (domain) used to authenticate the user ID specified by the **USER-ID** attribute. This is a logical name that represents the ABL code that handles the actual authentication of the user ID. You must set this value before sealing the object (logging in the client session). To be used for validating and setting user identities, this value must be identical to one of the following:  
  - The **Name** field setting for a domain entry created in a connected-database trusted domain registry using the **Authentication System Domains** dialog box in Data Administration (see the Data Administration online help).  
  - The **domain-name** parameter set for a domain entry created in the application trusted domain registry using the **REGISTER-DOMAIN( )** method of the **SECURITY-POLICY** system handle (see the “Managing OpenEdge session IDs” section on page 2–23.)  
  **Note:** It is up to you to ensure that the domain name you specify actually corresponds to the authentication system that you used to authenticate the user ID. |
| **DOMAIN-DESCRIPTION** | A description of the authentication system (domain) used to authenticate the user’s ID. |
| **DOMAIN-TYPE**      | The type of the authentication system (domain) used to authenticate the user’s ID. |
| **LOGIN-EXPIRATION-TIMESTAMP** | A **DATETIME-TZ** value that specifies when the client-principal object (and any login session started with it) expires. When ABL detects that the specified time has passed, it sets the **LOGIN-STATE** attribute to "EXPIRED" and the client-principal object becomes unusable for validation and any started client login session for the object ends. If you do not set this value, ABL ignores it and leaves it set to the Unknown value (?). |
| **AUDIT-EVENT-CONTEXT** | When auditing is enabled, an application-defined value that is stored in the _Event-context field of each audit event record generated by a client-principal object **SEAL( )**, **LOGOUT( )**, or **AUTHENTICATION-FAILED( )** method. |
| **CLIENT-TTY**       | The name of the terminal display for the user’s login session. |
| **CLIENT-WORKSTATION** | The name of the host workstation on which the user is working. |
Client-principal object methods

After you create a client principal object, you can set and read any number of application-defined properties using methods on the object handle. After all attributes and properties have been set, you can invoke additional methods to start and manage a client login session. Table 2–4 describes the methods that you can invoke on a client-principal object, listed in general usage order. For more information on managing client login sessions with these methods, see the “Managing client login sessions” section on page 2–16.

Table 2–4: Client-principal object methods

<table>
<thead>
<tr>
<th>This method . . .</th>
<th>Provides this function . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Property set and get methods</strong></td>
<td></td>
</tr>
<tr>
<td>\textbf{SET-PROPERTY( } \textit{property-name, property-value } \textbf{)}</td>
<td>Defines a property in the client-principal object with a name \textit{(property-name)} specified by a quoted string and a value specified by a character expression \textit{(property-value)}. You can call this method only once for each property you define in the object.</td>
</tr>
<tr>
<td>\textbf{LIST-PROPERTY-NAMES( )}</td>
<td>Returns a comma-separated list of all application-defined properties set using the \textbf{SET-PROPERTY( )} method.</td>
</tr>
<tr>
<td>\textbf{GET-PROPERTY( } \textit{property-name } \textbf{)}</td>
<td>Given the name of an application-defined property \textit{(property-name, expressed as a quoted string)}, returns the value assigned to that property using the \textbf{SET-PROPERTY( )} method.</td>
</tr>
<tr>
<td><strong>Login session management methods</strong></td>
<td></td>
</tr>
<tr>
<td>\textbf{AUTHENTICATION-FAILED ( [ reason ] )}</td>
<td>Indicates that the user ID associated with an unsealed client-principal object cannot be authenticated. After calling this method, the client-principal object is invalidated, its attributes and properties cannot be changed, and the object cannot be sealed. Calling this method also sets the LOGIN-STATE attribute to &quot;FAILED&quot; and writes the value of any character expression you provide for the \textit{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.</td>
</tr>
</tbody>
</table>
Given a character expression (`key`), indicates that the user ID associated with an unsealed client-principal object has successfully logged into its client login session. This method also generates an optional client login session record in connected databases. Before calling this method, all of the required attributes must be set (see Table 2–3).

Calling this method seals the client-principal object so you cannot set any new or existing attributes or properties for the object. It also 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 `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 `key` is identical to the access code/domain key of an authentication domain stored in a trusted domain registry (see the “Managing client login sessions” section on page 2–16).

Calling this method also sets the `SEAL-TIMESTAMP` attribute and checks the value of the `LOGIN-EXPIRATION-TIMESTAMP` attribute (see Table 2–3). If the client-principal object has expired, ABL sets the `LOGIN-STATE` attribute to "EXPIRED". Otherwise, it sets the attribute to "LOGIN".

If auditing is enabled and auditing policies permit, the method generates an audit event record, along with any optional client login session record, to indicate a valid user login.

After calling this method, you can use the client-principal object to assert and validate user identities using the `SET-DB-CLIENT` function or the `SET-CLIENT( )` method on the `SECURITY-POLICY` system handle.

Conversely, this method allows you to transport the user identity represented by a client-principal object from one ABL session to another.

<table>
<thead>
<tr>
<th>This method...</th>
<th>Provides this function...</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEAL( key )</td>
<td>Given a character expression (<code>key</code>), indicates that the user ID associated with an unsealed client-principal object has successfully logged into its client login session. This method also generates an optional client login session record in connected databases. Before calling this method, all of the required attributes must be set (see Table 2–3). Calling this method seals the client-principal object so you cannot set any new or existing attributes or properties for the object. It also 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 <code>key</code> 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 <code>key</code> is identical to the access code/domain key of an authentication domain stored in a trusted domain registry (see the “Managing client login sessions” section on page 2–16). Calling this method also sets the <code>SEAL-TIMESTAMP</code> attribute and checks the value of the <code>LOGIN-EXPIRATION-TIMESTAMP</code> attribute (see Table 2–3). If the client-principal object has expired, ABL sets the <code>LOGIN-STATE</code> attribute to &quot;EXPIRED&quot;. Otherwise, it sets the attribute to &quot;LOGIN&quot;. If auditing is enabled and auditing policies permit, the method generates an audit event record, along with any optional client login session record, to indicate a valid user login. After calling this method, you can use the client-principal object to assert and validate user identities using the <code>SET-DB-CLIENT</code> function or the <code>SET-CLIENT( )</code> method on the <code>SECURITY-POLICY</code> system handle.</td>
</tr>
</tbody>
</table>

| EXPORT-PRINCIPAL( ) | Converts and returns a sealed (logged in) 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 parameter to an AppServer session to set another client-principal object using the `IMPORT-PRINCIPAL` method. Thus, this method allows you to transport the user identity represented by a client-principal object from one ABL session to another. |
Managing identities

Table 2–4: Client-principal object methods

<table>
<thead>
<tr>
<th>This method . . .</th>
<th>Provides this function . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPORT-PRINCIPAL( expression )</td>
<td>Given a RAW expression that represents a sealed client-principal object (as output from the EXPORT-PRINCIPAL( ) method), sets a target client-principal object to the contents and state of the sealed and exported client-principal object. You can then use the new sealed client-principal object to set user identities using the SET-DB-CLIENT function or the SET-CLIENT( ) method on the SECURITY-POLICY system handle.</td>
</tr>
</tbody>
</table>
| VALIDATE-SEAL( [ key ] ) | 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 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 an access code/domain key value in one of the following ways:
- Pass it a parameter (key) whose value is identical to the key value used to seal the object.
- Ensure that the application trusted domain registry contains an entry for the authentication domain defined for the client-principal object (DOMAIN-NAME attribute) and that the entry includes an access code whose value is identical to the key value used to seal the object. (see the “Managing OpenEdge session IDs” section on page 2–23.)

Calling this method also checks the value of the LOGIN-EXPIRATION-TIMESTAMP attribute (see Table 2–3). If the client-principal object has expired, ABL sets the LOGIN-STATE attribute to "EXPIRED". |
| LOGOUT( ) | Indicates that the user ID associated with a sealed client-principal object has logged out of its client login session. After calling this method, the client-principal object is invalidated, its attributes and properties cannot be changed, and the object cannot be sealed.

Calling this method also sets the LOGIN-STATE attribute to "LOGOUT".

If auditing is enabled and auditing policies permit, the method generates an audit event record to indicate a user logout. |
Managing client login sessions

A given ABL session can create and maintain multiple client-principal objects, and hence multiple client login sessions for different identities. These login sessions then each function according to the particular identity they represent and selected database options.

Requirements for using client login sessions

The common requirement for creating client login sessions for all types of user identities is to initialize some required data values for the session, including (among others) an authenticated user ID. You can authenticate this ID at run time using any acceptable method, including the OpenEdge internal authentication system (\_User table) or any external authentication system that you can access from ABL. For more information, see the “Authenticating user identities” section on page 2–3.

You set these required data values using attributes and methods of a client-principal object. You must complete the initialization of these values and seal the client-principal object in order to begin the client login session. Sealing a client-principal object prevents any further changes to its initialized data values and allows it to be used for validating and setting user identities.

If you use an external authentication system, you must also create a corresponding domain entry in the appropriate trusted domain registry to represent the ABL code that handles the authentication:

- The database trusted domain registry for each connected OpenEdge RDBMS where you set a database connection ID
- The application trusted domain registry for each ABL session where you set an OpenEdge session ID

This domain entry must be available in the appropriate trusted domain registry before you use the client-principal object to validate any user ID that has been authenticated using the corresponding authentication system. For a database trusted domain registry, you can create this entry at database configuration time, before you run the application. For an application trusted domain registry, you must create the entry at application run time when you build the registry for the ABL session. Optionally, a database can also be configured to trust the application domain registry, instead of its own database domain registry, to validate its database connection ID.

Your application might rely on multiple external authentication systems to authenticate user IDs. However, to be acceptable for asserting a user identity, a given authentication system must have a corresponding and enabled domain entry registered in the trusted domain registry that you use to assert the user identity.

Each domain entry requires an access code (sometimes referred to as a domain key). An access code is a the password-like value that you set for the Access Code field of the Authentication System Domains dialog box in Data Administration for your database’s trusted domain registry, or that you pass as a parameter to create a domain entry in your application’s trusted domain registry. For each enabled domain entry used to validate a user identity, your application needs access to the corresponding access code value in order to seal the client-principal object for validation. You can input this value at run time or otherwise hide it somewhere within your application code.
For more information on trusted domain registries, see *OpenEdge Getting Started: Core Business Services*. For more information on configuring database domain registries, see the Data Administration online help or *OpenEdge Development: Basic Database Tools*. For information on building an application trusted domain registry, see the “Managing OpenEdge session IDs” section on page 2–23 in this manual.

If you want the client 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 client login session that you successfully start, OpenEdge generates a single `_client-session` record keyed on the client login session ID.

**Note:** The client login session ID, which uniquely identifies a client login session, is different from the OpenEdge session ID, which is a user identity. Multiple unique client login session records can exist for the same user identity, whether the identity is the OpenEdge session ID or a database connection ID.

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 *OpenEdge Getting Started: Core Business Services*.

**Note:** If the database is also audit-enabled, the recorded client login session ID identifies an audit event context used to record several client login session audit events. For more information on using audit event context, see Chapter 3, “Auditing.”

### Starting and maintaining client login sessions

You can start and manage client 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 client login session:

1. Create a client-principal object using the **CREATE CLIENT-PRINCIPAL** statement.

2. Initialize all required, and any optional, attributes of the client-principal object handle and also any application-defined properties that you want to associate with the client login session using the **SET-PROPERTY( )** method on the client-principal object handle. For more information on these attributes, see Table 2–3. For more information on the **SET-PROPERTY( )** method, see Table 2–4.

3. Authenticate the user ID that you set for the client-principal object.

4. If the user ID that you set fails to authenticate, you can call the **AUTHENTICATION-FAILED( )** method on the client-principal object handle (see Table 2–4) to freeze all settings for the client-principal object. At this point, you cannot create a client login session with the specified client-principal object, and you must skip the remaining steps for using it.
5. When you have initialized all the client-principal object attributes that you need for a given authenticated user ID, login the client session by calling the SEAL( ) method on the client-principal object handle (see Table 2–4), which seals the client-principal object against any further changes to its data values. It also sets the read-only SEAL-TIMESTAMP attribute (see Table 2–2), which records the effective user authentication time for the client login session.

6. Given that the login is successful in Step 5, you can then use the client-principal object to assert and validate a user identity for the client login session. For more information, see:

   - The “Managing database connection IDs” section on page 2–19
   - The “Managing OpenEdge session IDs” section on page 2–23

Note: You might login the client session after a database connection ID has already been asserted and established using the OpenEdge internal authentication system. In this case, you skip Step 6.

7. If you need to transport the client login session between (or even within) ABL sessions, 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 2–4). For more information on when you might need to do this, see the “Managing application user IDs in n-tier applications” section on page 2–28.

8. To invalidate a client 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 2–4). 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 client login session, unless you set the LOGIN-EXPIRATION-TIMESTAMP attribute on the client-principal object handle (see Table 2–3), before you seal the client-principal object in Step 5.

Note: After you no longer need a client-principal object, be sure to delete it using the DELETE OBJECT statement.

At any time during initialization and management of a client login session you can query the client-principal object attributes directly and any properties you have defined using the GET-PROPERTY( ) method (see Table 2–3). Attributes you might want to query for the state of the client login session include the read-only LOGIN-STATE and STATE-DETAIL attributes (see Table 2–2), which reflect changes in the login session and the usability of the associated client-principal object.
Managing identities

Managing database connection IDs

By default, OpenEdge authorizes all permissions for database and auditing operations using the database connection ID established for each connected OpenEdge RDBMS. You can set the database connection ID for a database from two basic sources:

1. A user ID authenticated using the OpenEdge internal authentication system (\_User table)
2. A user ID authenticated using an external authentication system

If you do not set a database connection ID for a database, the database connection ID is considered blank. You can also clear an existing database connection ID, which resets the effective ID to blank.

Using the internal authentication system to set database connection IDs

Any user ID that you authenticate using the \_User table becomes the database connection ID by default, whether you authenticate it using the User ID (-U) and Password (-P) connection parameters in the ABL CONNECT statement or using the SETUSERID function. If you do nothing else, the database where you set this connection ID authorizes all of its operations using this database connection ID.

However, you can also add a client login session to your application for this database connection ID by creating and initializing a client-principal object for it (see the “Managing client login sessions” section on page 2–16). If you only require that this database connection ID be used in the OpenEdge session where it is authenticated, you do not need to further assert and validate the identity associated with this client-principal object beyond logging in and sealing the client principal object using the SEAL( ) method with any access code value that you choose. If your database options are set to record client login sessions, this is sufficient to record the client login session in the database, and also to provide login session context for auditing (if enabled).
Thus, you might use a code fragment such as this, which sets up an internally authenticated
client login session and sets the database connection ID for an inventory database (already
connected). The bold-faced code features the client login session management and
authentication elements, as follows:

**Login session for an internally authenticated database connection ID**

```ecl
DEFINE VARIABLE cPassword AS CHARACTER NO-UNDO.
DEFINE VARIABLE cUserID AS CHARACTER NO-UNDO.
DEFINE VARIABLE hCP AS HANDLE NO-UNDO.

/* Get user ID and password */
cUserID = ...
cPassword = ...

/* Create and initialize client-principal for use with Inventory database */
CREATE CLIENT-PRINCIPLE hCP.
ASSIGN
  hCP:SESSION-ID = "Inventory:" + cUserID + STRING(NOW)
  hCP:USER-ID = cUserID
  hCP:DOMAIN-NAME = "InventoryDB".

/* Authenticate and login database connection ID session for the single
database connection */
IF SETUSERID(cUserID, cPassword) THEN
  hCP:SEAL("AlwaysValid").
ELSE
  hCP:AUTHENTICATION-FAILED("User not authenticated.").

/* Handle results of authentication */
IF hCP:LOGIN-STATE = "LOGIN" THEN DO: /* Do Inventory stuff... */
  ...
  hCP:LOGOUT().
  DELETE OBJECT hCP.
END.
ELSE DO:
  RETURN hCP:LOGIN-STATE + ": " + hCP:STATE-DETAIL.
  DELETE OBJECT hCP.
END.
```

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 to set the
SESSION-ID attribute. For more information, see the “Creating and managing unique object
identities” section on page 2–36.

Also, in this fragment, the access code value ("AlwaysValid") that you use to seal the
client-principal object can be anything. Since you are not validating your database connection
ID in the client login session against a trusted domain registry and the login session does not
survive the OpenEdge session, the security risk is minimal. In this case, you are likely creating
the login session only to record its context.

If you do have reason to validate the database connection ID using the client-principal object,
you must register a domain entry for the _User table in the OpenEdge RDBMS’s trusted domain
registry and manage the client-principal object as if it represents an externally authenticated user
identity (see the “Setting a database connection ID from an external authentication system”
section on page 2–21). A common reason to do this is to transport and use the same login session
information between multiple OpenEdge sessions within your application environment (see the
“Managing application user IDs in n-tier applications” section on page 2–28).
Setting a database connection ID from an external authentication system

If you choose to set the database connection ID from a user ID that you authenticate externally, you must have a corresponding domain entry registered in a trusted domain registry. By default, this registry must be the OpenEdge RDBMS’s trusted domain registry.

**Note:** You can set multiple database connection IDs using a single client-principal object by configuring database options to have each database trust the application domain registry. In this case, the database connection ID essentially is and is managed exactly as a OpenEdge session ID (see the “Managing OpenEdge session IDs” section on page 2–23). For more information on setting database options to trust the application domain registry, see the sections on synchronizing user identities in *OpenEdge Getting Started: Core Business Services* and the online help for setting database options in Data Administration.

Any user ID that you authenticate with an external authentication system has no OpenEdge-supported functional identity until you specifically assert and validate that identity against the corresponding authentication domain. This you must do using a client-principal object.

**To assert and validate an externally authenticated database connection ID:**

1. Input the candidate user ID and authentication criteria (password, for example).
2. Create and initialize a client-principal object with the user ID and corresponding authentication domain information that is in your database trusted domain registry.
3. Authenticate the user ID through the external authentication system.
4. If the authentication succeeds, proceed to Step 5. If the authentication fails, invalidate the client-principal object with the `AUTHENTICATION-FAILED()` method and proceed or exit the procedure accordingly.
5. Log in the client session and seal the client-principal object using the access code assigned to your authentication domain.
6. Assert and validate the authenticated user ID represented by the client-principal object as the database connection ID for your database (or databases) using the ABL SET-DB-CLIENT function. The SET-DB-CLIENT function validates the client-principal object against a database trusted domain registry in order to set the corresponding database connection ID. Optionally, this can be the application domain registry, instead of the database’s own domain registry. For more information, see the “Managing OpenEdge session IDs” section on page 2–23.
7. Manage the rest of the client login session as usual. For more information, see the “Managing client login sessions” section on page 2–16.
Thus, you might use a code fragment such as this, which sets up an externally authenticated client login session in order to set a database connection ID for an inventory database (already connected). The bold-faced code features the client login session management and authentication elements, as shown:

**Login session for an externally authenticated database connection ID**

```plaintext
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.

/* 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 ID. This value, returned using the getAccessCode user-defined function, must match the access code configured for the authentication domain entry in the OpenEdge RDBMS’s 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 to set the SESSION-ID attribute. For more information, see the “Creating and managing unique object identities” section on page 2–36.

**Managing OpenEdge session IDs**

An OpenEdge session ID can serve a variety of functions, including any one or more of the following:

- Serving as a single session-wide and database-independent identity for authorizing application-defined features
- Serving as a single session-wide database connection ID for all database connections in an OpenEdge session (see the “Managing database connection IDs” section on page 2–19)
- Serving as a session-wide auditing ID (see the “Managing auditing IDs” section on page 2–36)

You can set the OpenEdge session ID from a user ID that is authenticated using an external authentication system. If you do not explicitly set it, the OpenEdge session ID is considered blank. You can also clear an existing OpenEdge session ID, which resets the effective ID to blank.

To set an OpenEdge session ID, you must build a trusted domain registry for the application at run time in order to assert and validate an authenticated user ID as a valid OpenEdge session ID, and optionally as a database connection ID.

**Building the application trusted domain registry**

You can build an application trusted domain registry from the contents of an existing database trusted domain registry or create individual domain entries directly in your ABL code. The SECURITY-POLICY system handle provides methods that you can use to build an application trusted domain registry as described in Table 2–5.
### Table 2–5: Methods for building an application domain registry

<table>
<thead>
<tr>
<th>This SECURITY-POLICY method...</th>
<th>Provides this function...</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LOAD-DOMAINS</strong> (<strong>integer-expression</strong></td>
<td>Loads the application trusted domain registry with the existing domain entries from the database trusted domain registry of a specified OpenEdge RDBMS. (The <strong>integer-expression</strong> parameter specifies the database by its order of connection in the OpenEdge session.)</td>
</tr>
<tr>
<td><strong>logical-name</strong></td>
<td>You can only call this method once successfully. A successful call to this method implicitly locks the application trusted domain registry against any further changes.¹</td>
</tr>
<tr>
<td><strong>alias</strong></td>
<td><strong>Note:</strong> This method for loading the application trusted domain registry is more secure than using the <strong>REGISTER-DOMAIN( )</strong> and <strong>LOCK-REGISTRATION( )</strong> methods.</td>
</tr>
<tr>
<td><strong>REGISTER-DOMAIN</strong> (<strong>domain-name</strong>, <strong>access-code</strong></td>
<td>Creates a new domain entry in the application trusted domain registry. The <strong>domain-name</strong> parameter specifies the required value for the <strong>DOMAIN-NAME</strong> attribute of the client-principal object in order for that object to validate against the domain entry (see Table 2–3). The <strong>access-code</strong> parameter specifies the value to use as the key for sealing a client-principal object in order for the object to validate against this domain entry (see Table 2–4).</td>
</tr>
<tr>
<td><strong>[, domain-description</strong></td>
<td>You can no longer call this method after calling the <strong>LOCK-REGISTRATION( )</strong> method. <strong>Note:</strong> The <strong>domain-description</strong> and <strong>domain-type</strong> values do not have to match the <strong>DOMAIN-DESCRIPTION</strong> and <strong>DOMAIN-TYPE</strong> attributes in a client-principal object in order for the object to validate against the domain entry.</td>
</tr>
<tr>
<td><strong>[, domain-type ]]</strong></td>
<td></td>
</tr>
<tr>
<td><strong>LOCK-REGISTRATION( )</strong></td>
<td>Locks the application trusted domain registry against any further registrations using the <strong>REGISTER-DOMAIN( )</strong> method.¹</td>
</tr>
<tr>
<td></td>
<td>You must call this method before you can use the application domain registry to validate identities that you have built using the <strong>REGISTER-DOMAIN( )</strong> method.</td>
</tr>
</tbody>
</table>

¹. Once the application trusted domain registry is locked, it cannot be changed during the OpenEdge session. The only way to change it is to restart the session and reload the registry.
The REGISTER-DOMAIN( ) method provides the flexibility to dynamically define a trusted authentication domain not otherwise available in a database domain registry. However, until you call the LOCK-REGISTRATION( ) method, you have risk of a security breach by allowing the possible registration of rogue domains in the application domain registry. The LOAD-DOMAINS( ) method minimizes this risk by providing a secure pathway to load and lock the entire contents of an existing database domain registry (and only the contents of that registry) into the application domain registry.

Setting the OpenEdge session ID

Most ABL applications authenticate a user ID that is authenticated externally, without any reference to the _User table. This session-wide user ID is then used to authorize application features using the ABL CAN-DO function (see the “Authorizing access to procedures and database resources” section on page 2–39) or some other application-defined authorization mechanism.

Using a client-principal object and the application trusted domain registry with appropriate database configuration options, you can assert this externally authenticated user ID to OpenEdge as the OpenEdge session ID, which can then recognize it for use with various OpenEdge features. For example:

- If you select the database option to trust the application domain registry, the database recognizes the OpenEdge session ID as its database connection ID. In this way, the database can authorize access to its tables and fields based on the effective OpenEdge session ID. If all connected OpenEdge databases are configured to trust the application domain registry, you can efficiently set all of their database connection IDs to this single OpenEdge session ID. Thus, you can use the OpenEdge session ID to synchronize all user identities for the entire OpenEdge session.

- If you select the database option to use the application user ID as the auditing ID, the database records the OpenEdge session ID as the auditing ID in its audit event records.

Note: Set the database option to use the application user ID as the auditing ID only if you are individually managing the database connection ID for each database (see the “Managing database connection IDs” section on page 2–19). If you synchronize all database connection IDs to the OpenEdge session ID, all connected databases use the OpenEdge session ID as the auditing ID by default.

With a common application user ID, you can set a single OpenEdge session ID maintained using a client-principal object and transport this user identity between OpenEdge sessions of a distributed application to maintain a consistent identity among them (see the “Managing application user IDs in n-tier applications” section on page 2–28).

If you want to set an OpenEdge session ID from a user ID that you authenticate externally, you must have a corresponding domain entry registered in the application trusted domain registry. As described in an earlier section (see the “Building the application trusted domain registry” section on page 2–23), you can build this registry within the OpenEdge session at run time entirely from a configured database trusted 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 application domain registry from the existing database domain registry of an OpenEdge RDBMS.

Any user ID that you authenticate with an external authentication system has no OpenEdge-supported functional identity until you specifically assert and validate that identity against the corresponding authentication domain. This you must do using a client-principal object.
To assert and validate an externally authenticated OpenEdge session ID:

1. Build the application trusted domain registry using methods of the SECURITY-POLICY system handle.

2. Input the candidate user ID and authentication criteria (password, for example).

3. Create and initialize a client-principal object with the user ID and corresponding authentication domain information that is in your application trusted domain registry.

4. Authenticate the user ID 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 proceed or exit the procedure accordingly.

6. Log in the client session and seal the client-principal object using the access code assigned to your authentication domain.

7. Assert and validate the authenticated user ID represented by the client-principal object as an OpenEdge session ID using the SET-CLIENT( ) method on the SECURITY-POLICY system handle. The SET-CLIENT( ) method always validates the client-principal object against the application trusted domain registry in order to set the OpenEdge session ID.

   By default, if any connected databases do not currently have database connection IDs, the SET-CLIENT( ) method also attempts to validate and set the database connection ID for each of these databases using the trusted domain registry configured for each database. This setting can always be overridden using the ABL SETUSERID or SET-DB-CLIENT functions. However, if database options are set to trust the application domain registry, both SET-CLIENT( ) and SET-DB-CLIENT use the application trusted domain registry to validate and set the database connection ID for that database.

   **Note:** If the application trusted domain registry is not initialized and remains unlocked, SET-CLIENT( ) does not validate an OpenEdge session ID, but does propagate the client-principal object validate and set database connection IDs, according to database connection status and configuration.

8. Manage the rest of the client login session as usual (see the “Managing client login sessions” section on page 2–16).
Thus, you might use a code fragment such as the following, which sets up an externally authenticated client login session in order to set an OpenEdge session ID for an inventory application that accesses a single database (already connected). The bold-faced code features the client login session management and authentication elements, as shown:

### Login session for an externally authenticated OpenEdge session ID

```plaintext
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.
```

2–27
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 ID. This
value, returned using the getAccessCode user-defined function, must match the access code
configured for the authentication domain entry in the application’s 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 to set the
SESSION-ID attribute. For more information, see the “Creating and managing unique object
identities” section on page 2–36.

Note: If no database connection ID is set for the connected database, the SET-CLIENT( )
method in this fragment also sets the database connection ID, 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 application domain registry
(which in this example has the same content as the database domain registry) to set the
database connection ID. 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 ID, and the OpenEdge session ID is blank.

Managing application user IDs in n-tier applications

An application user ID exists essentially to specify a single user identity that is shared between
client and AppServer sessions of an n-tier application. The application user ID typically
originates from the client and is used to set the OpenEdge session ID in each AppServer session.
It might also be used for synchronizing database connection IDs in each AppServer session, as
appropriate.

In n-tier applications, the problem of managing application user IDs consists of these main
issues:

• How to transport the application user ID between the AppServer session and a client
  session

• How to share a single login for an application user ID across multiple requests to an
  AppServer running in stateless or state-free operating mode

Note: For information on AppServer operating modes and how AppServer and client
sessions interact with them, see OpenEdge Application Server: Developing AppServer
Applications.

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 client
  login (state-free)

• Share a single client-principal object in a common user context storage to assert the user’s
  identity on each agent involved in a given client’s requests

• Logout and destroy 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 an n-tier application, the AppServer’s ABL code creates and maintains the client-principal object on behalf of the client and exchanges identity information, based on the operating mode, as described in Table 2–6.

### Table 2–6: Identity management for n-tier applications

<table>
<thead>
<tr>
<th>In this operating mode . . .</th>
<th>The AppServer . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>State-aware or state-reset</td>
<td>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.</td>
</tr>
<tr>
<td>Stateless</td>
<td>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 client login sessions, and use the SERVER-CONNECTION-ID on the SESSION system handle to key access to the client-principal object in user context storage.</td>
</tr>
<tr>
<td>State-free</td>
<td>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. The AppServer creates this identifier for each client-principal object that it creates and communicates it as a security token to the client in response to its initial login request to the AppServer. The client must then return this security token with each request to the AppServer, so the AppServer agent can retrieve the shared client-principal object from user context storage.</td>
</tr>
</tbody>
</table>
Initializing a client-principal object for an n-tier application

The following code fragment shows how you might initialize a client-principal object to account for the type of OpenEdge session environment it is used in:

Client-principal object initialization for n-tier applications

```
CREATE CLIENT-PRINCIPAL m_hCP.
ASSIGN
    m_hCP:USER-ID = p_cUserId
    m_hCP:DOMAIN-NAME = p_cAuthDomain.
IF (SESSION:REMOTE) THEN DO: /* AppServer session */
    IF (SESSION:SERVER-OPERATING-MODE = "State-free") THEN DO:
        cSessionID = BASE64-ENCODE(GENERATE-UUID).
        m_hCP:SESSION-ID = SUBSTRING(cSessionID, 1, 22)
    END.
    ELSE/* Session-managed operating modes */
    m_hCP:SESSION-ID = SESSION:SERVER-CONNECTION-ID.
END.
ELSE DO: /* Client session */
    cSessionID = BASE64-ENCODE(GENERATE-UUID).
    m_hCP:SESSION-ID = SUBSTRING(cSessionID, 1, 22)
END.

m_hCP:AUDIT-EVENT-CONTEXT = p_cUserId + "@" + p_cAuthDomain.
m_hCP:CLIENT-TTY = SESSION:CLIENT-TYPE + "." + SESSION:DISPLAY-TYPE
```

For any of the session-managed operating modes, the fragment sets the SESSION-ID attribute to the SERVER-CONNECTION-ID attribute on the SESSION system handle. This value uniquely identifies the client connection for a given AppServer session and is especially useful for an AppServer session running in stateless operating mode. Since a stateless application service can identify each request by the client connection, it can use this client connection identifier for the client-principal object that represents a client identity. However, for stateless auditing purposes, you might also use the SERVER-CONNECTION-ID value to key references to a shared client-principal object, but set the SESSION-ID attribute to a universal unique identifier (UUI) using the ABL GENERATE-UUID function. This value ensures that a unique client login session auditing record across all possible AppServer sessions.

For a state-free AppServer operating mode, the fragment sets the SESSION-ID attribute to a universal unique identifier (UUID), which is generated using the ABL GENERATE-UUID function and encoded as Base64 using the ABL BASE64-ENCODE function. For more information on these functions, see the “Creating and managing unique object identities” section on page 2–36. Since clients of a state-free AppServer make no connections to the AppServer, the application service needs to maintain its own unique identifiers for the client-principal objects that represent client identities.

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, but they must do it differently according to their respective session-managed and session-free behavior.
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.

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 application trusted 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:

```sql
DEFINE VARIABLE cAccessCode AS CHARACTER NO-UNDO.
DEFINE 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, seal, and login client-principal object */
...

cSessionID = BASE64-ENCODE(GENERATE-UUID).
hCP:SESSION-ID = SUBSTRING(cSessionID, 1, 22)
hCP:SEAL(cAccessCode).

/* Export client-principal 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-principal 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 OpenEdge session ID 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:

```plaintext
/* Lookup client-principal object and remove from context database */
FIND UserContext
   WHERE Principal-key = SESSION:SERVER-CONNECTION-ID NO-ERROR.
DELETE UserContext NO-ERROR.
...  
```

Note: The client-principal object is already imported using the activation procedure and its identity will be cleared using the deactivation procedure.

7. In the AppServer’s configured shutdown procedure, empty the user context database, for example:

```plaintext
/* 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 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. Following is a typical procedure for managing state-free user identities.

To manage user identities for a state-free application service:

1. In the AppServer’s configured startup procedure, build any required application trusted 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 set a unique 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-ID attribute value.

d. Return the SESSION-ID attribute value as an output parameter to the client.

For example:

```c
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, seal, and login client-principal object */
... cSessionID = BASE64-ENCODE(GENERATE-UUID).
hCP:SESSION-ID = SUBSTRING(cSessionID, 1, 22)
hCP:SEAL(cAccessCode).

/* Export client-principal 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 security token to client */
cSessionID = hCP:SESSION-ID.
...
```

3. In all non-login procedures for the application service:

   a. Pass in security token as input parameter.

   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 OpenEdge session ID for the next user.
For example:

```plaintext
DEFINE INPUT PARAMETER cSessionID AS CHARACTER NO-UNDO.
DEFINE VARIABLE rCP AS RAW NO-UNDO.
DEFINE SHARED VARIABLE hCP AS HANDLE NO-UNDO.

CREATE CLIENT-PRINCIPAL hCP.

/* Lookup client-principal object in context database */
FIND UserContext WHERE Principal-key = cSessionID NO-ERROR.
rCP = UserContext.Principal.

/* Import client-principal 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 ID for next user */
hCP:LOGOUT().
DELETE OBJECT hCP.
...
```

4. In the application service’s logout procedure:
   a. Pass in security token as input parameter.
   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:

```plaintext
DEFINE INPUT PARAMETER cSessionID AS CHARACTER NO-UNDO.

/* 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:

```plaintext
/* Empty the context database */
FOR EACH UserContext:
   DELETE UserContext NO-ERROR.
END.
...
```
Managing auditing IDs

By default, OpenEdge uses the database connection ID as the auditing ID. As described in earlier sections, OpenEdge can otherwise take the auditing ID for a given database from the OpenEdge session ID, depending on a setting in database auditing options. No matter what source you use for the auditing ID, you might note the following points:

- The auditing ID only specifies 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 ID for each database always controls database access and auditing privileges.

- If you configure the database option to trust the application domain registry, this causes both the ABL SET-DB-CLIENT function and the SET-CLIENT( ) method on the SECURITY-POLICY system handle to assert and validate the database connection ID using the application’s trusted domain registry (instead of using the database’s own trusted domain registry). If you use SET-CLIENT( ) with this database option to set the database connection ID, which also asserts and validates the OpenEdge session ID, the effective auditing ID is identical to the OpenEdge session ID, regardless of any setting database auditing options that you set.

For more information on managing the auditing ID within an OpenEdge session, see Chapter 3, “Auditing.”

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 client 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 the “Managing application user IDs in n-tier applications” section on page 2–28).

OpenEdge provides two ABL functions that work specifically with unique values, as described in Table 2–7.

Table 2–7: Unique value functions

<table>
<thead>
<tr>
<th>This ABL function . . .</th>
<th>Returns this value . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>GENERATE-UUID</td>
<td>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.</td>
</tr>
<tr>
<td>GUID( [ UUID ] )</td>
<td>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.</td>
</tr>
</tbody>
</table>
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
- Hexadecimal

Table 2–8 describes these functions.

### Table 2–8: Data encoding/decoding functions

<table>
<thead>
<tr>
<th>This ABL function . . .</th>
<th>Returns this value . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASE64-ENCODE( expression )</td>
<td>A LONGCHAR value consisting of a Base64 representation of the RAW or MEMPTR value passed as expression</td>
</tr>
<tr>
<td>BASE64-DECODE( expression )</td>
<td>A MEMPTR value that represents the CHARACTER or LONGCHAR value containing a Base64 string passed as expression</td>
</tr>
<tr>
<td>HEX-ENCODE( expression )</td>
<td>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</td>
</tr>
<tr>
<td>HEX-DECODE( expression )</td>
<td>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</td>
</tr>
</tbody>
</table>

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 the “Managing and transporting crypto data” section on page 2–61.

### Using secure database connections

You can use the Secure Sockets Layer (SSL) to provide a security infrastructure that protects communications between a database client and server. SSL 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 (SSL client) and the responder (SSL server).
The server is identified by the private key that it stores and the client is identified as a valid SSL client for that server by the public key that it stores and provides to the server. SSL 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 SSL server.

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

To secure a connection to the database using SSL:

1. Ensure that your OpenEdge RDBMS is SSL-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 SSL 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 SSL on the database server.

3. Using the `CONNECT()` statement, connect to the database using the client connection parameters required to establish an SSL connection, as shown in Table 2–9.

### Table 2–9: SSL database connection parameters

<table>
<thead>
<tr>
<th>Connection parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SSL-based Connection (-ssl)</td>
<td>Specifies a Secure Socket Layer connection. (Required)</td>
</tr>
<tr>
<td>No Host Verify (-nohostverify)</td>
<td>Turns off host verification for a Secure Sockets Layer (SSL) connection.</td>
</tr>
<tr>
<td></td>
<td>(Optional)</td>
</tr>
<tr>
<td>No Session Reuse (-nosessionreuse)</td>
<td>Avoids the reuse of a Secure Sockets Layer (SSL) session ID. (Optional)</td>
</tr>
</tbody>
</table>

For more information on:

- OpenEdge SSL support and how to manage private and public keys and certificates to secure connections, see *OpenEdge Getting Started: Core Business Services*

- How to enable a database server for SSL connections, see *OpenEdge Data Management: Database Administration*

- Using the `CONNECT()` method to connect to a database, see Chapter 1, “Database Access” in this manual

- The database server startup and connection parameters used to enable SSL, see *OpenEdge Deployment: Startup Command and Parameter Reference*
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 IDs 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
- Defining activities-based user authorization
- Authorizing user access to tables and fields

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 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:

```
Maintenance and reporting

MAIN MENU
1) Add a new Customer
2) Change customer information
3) Display orders
4) Create mailing labels
5) Delete customer
6) EXIT

Selection: __
```

Enter data or press F 4 to end.

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 _adcust.p procedure allows you to enter Customer information.

### _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 _adcus2.p to authorize access based on the database connection ID for single connected database:

### _adcus2.p

```
IF NOT CAN-DO("manager , salesrep")
THEN DO:
   MESSAGE "You are not authorized to run this procedure"
   RETURN.
END.

REPEAT:
   INSERT customer WITH 2 COLUMNS.
END.
```

The first part of _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 ID 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. Table 2–10 lists the types of tokens you can specify.

Table 2–10: Values to use for ID lists

<table>
<thead>
<tr>
<th>Value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>All users are allowed access</td>
</tr>
<tr>
<td>user</td>
<td>This user has access</td>
</tr>
<tr>
<td>!user</td>
<td>This user does not have access</td>
</tr>
<tr>
<td>string*</td>
<td>Users whose IDs begin with “string” have access</td>
</tr>
<tr>
<td>!string*</td>
<td>Users whose IDs begin with “string” do not have access</td>
</tr>
</tbody>
</table>

For more information on the CAN–DO function, see OpenEdge Development: 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.

```
IF USERID <> "manager"
THEN DO:
  MESSAGE "You are not authorized to run this procedure ."
  RETURN.
END.
REPEAT:
  INSERT customer WITH 2 COLUMNS.
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 the “Defining activities-based user authorization” section on page 2–42.

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 the “Authenticating user identities” section on page 2–3. 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:

```sql
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. This chapter calls 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 permissions.
END.
```

This procedure lets you add records of activities to the permissions table until you press END–ERROR. Figure 2–1 shows records of application activities that you can add to the permissions table.

<table>
<thead>
<tr>
<th>Activity</th>
<th>Can-Run</th>
</tr>
</thead>
<tbody>
<tr>
<td>p-adcust.p</td>
<td>manager, salesrep</td>
</tr>
<tr>
<td>p-chcust.p</td>
<td>manager, salesrep</td>
</tr>
<tr>
<td>p-delcus.p</td>
<td>manager</td>
</tr>
<tr>
<td>print</td>
<td>manager, inventory</td>
</tr>
</tbody>
</table>

**Figure 2–1: Sample activity permissions entries**

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.
Figure 2–2 shows what happens when the user with user ID manager runs the i-adcust.p procedure.

When the specified user ID and the permission defined in the i-adcust.p record in the permissions table match, the user can run the procedure. Figure 2–3 shows what happens when the user with user ID inventory tries to run i-adcust.p. Because there is no match, the procedure displays a message and the user cannot run the remaining code.

Figure 2–3: 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

```plaintext
/* 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

```plaintext
/* 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

```i
/* 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:
    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

```i
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 such an include file is i-chkpirm.i.
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. Figure 2–4 shows an example security record in the permissions table.

![Sample security record for activity permissions](image-url)

### Figure 2–4: 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**

```plaintext
/* 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.

Figure 2–5 summarizes the security process developed for procedures and functions in an application.

![Figure 2–5: Summary of run-time authorization process](image)

1. The developer creates a record in the database for each application activity, including security administration.
2. The developer assigns one or more authorized users for each activity, including security administration.
3. The security administrator maintains and updates permissions for procedures and activities.
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 *OpenEdge Deployment: Managing 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 methods do not apply to buffers associated with individual temp-tables.

Table 2–11 lists the CAN-* attributes that you can use to test these data permissions.

**Table 2–11: CAN-* attributes for testing run-time data access permissions**

<table>
<thead>
<tr>
<th>This attribute</th>
<th>Indicates if the user can . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAN-CREATE</td>
<td>Create records in the database table associated with the given buffer object</td>
</tr>
<tr>
<td>CAN-DELETE</td>
<td>Delete records in the database table associated with the given buffer object</td>
</tr>
<tr>
<td>CAN-READ</td>
<td>Read records or fields in the database table associated with the given buffer object or buffer-field object</td>
</tr>
<tr>
<td>CAN-WRITE</td>
<td>Write records or fields in the database table associated with the given buffer object or buffer-field object</td>
</tr>
</tbody>
</table>
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 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 *OpenEdge Getting Started: Core Business Services.*

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
- Generating encryption keys
- Implementing symmetric cryptography in ABL
- Using message digests in ABL
- Managing and transporting crypto data
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. Table 2–12 lists the attributes that you can read and set for this system handle. These attributes are both readable and setable unless otherwise indicated.

Table 2–12: SECURITY-POLICY system handle attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENCRYPTION-SALT</td>
<td>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. <strong>Note:</strong> Typically, you never need to set this value, and then only if the security specification for your application requires it.</td>
</tr>
<tr>
<td>PBE-HASH-ALGORITHM</td>
<td>A CHARACTER value that specifies the hashing algorithm to use for generating a PBE key, which you can specify as &quot;MD5&quot; or &quot;SHA-1&quot; (default: &quot;SHA-1&quot;). <strong>Note:</strong> Typically, you never need to set this value, and then only if the security specification for your application requires it.</td>
</tr>
<tr>
<td>PBE-KEY-ROUNDS</td>
<td>A positive INTEGER value that specifies the number of algorithm iterations to use for PBE key generation (default: 1000). <strong>Note:</strong> 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.</td>
</tr>
<tr>
<td>SYMMETRIC-ENCRYPTION-ALGORITHM</td>
<td>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: &quot;AES_CBC_128&quot;).</td>
</tr>
<tr>
<td>SYMMETRIC-ENCRYPTION-KEY</td>
<td>A write-only RAW value that specifies the symmetric key to use for encryption and decryption (default: none).</td>
</tr>
</tbody>
</table>
For information on OpenEdge-supported defaults and options for setting these attributes, see *OpenEdge Development: 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 the “Implementing symmetric cryptography in ABL” section on page 2–56) 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 *OpenEdge Getting Started: Core Business Services*.

**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 the “Generating encryption keys” section on page 2–54.

**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 the “Using message digests in ABL” section on page 2–60). 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 Table 2–13.

### Table 2–13: Cryptographic key generation functions

<table>
<thead>
<tr>
<th>ABL function</th>
<th>Description</th>
</tr>
</thead>
</table>
| 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 the “Implementing symmetric cryptography in ABL” section on page 2–56. |
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 OpenEdge Getting Started: Core Business Services.

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 the “Using message digests in ABL” section on page 2–60. For more information on encrypting MAC keys for secure audit archiving, see the “Custom audit archiving tools” section on page 3–22.

**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 serious 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**

```
```

```
```

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 successfully 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 Table 2–14, whether you supply them as function options or as attributes of your application cryptography policy.

<table>
<thead>
<tr>
<th>Set this cryptographic parameter using . . .</th>
<th>This function option . . .</th>
<th>Or this <code>SECURITY-POLICY</code> handle attribute . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symmetric key</td>
<td><code>encrypt-key</code></td>
<td><code>SYMMETRIC-ENCRYPTION-KEY</code></td>
</tr>
<tr>
<td>Initialization vector</td>
<td><code>iv-value</code></td>
<td><code>SYMMETRIC-ENCRYPTION-IV</code></td>
</tr>
<tr>
<td>Algorithm, mode, and key size</td>
<td><code>algorithm</code></td>
<td><code>SYMMETRIC-ENCRYPTION-ALGORITHM</code></td>
</tr>
</tbody>
</table>

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 the “Creating and maintaining a cryptography policy” section on page 2–52
- Key generation, see the “Generating encryption keys” section on page 2–54
- Managing and transporting keys and data, see the “Managing and transporting crypto data” section on page 2–61
Example code using the ENCRYPT and DECRYPT functions

This section shows basic uses of the ABL symmetric cryptography functions and related ABL elements.

Example 2–1 shows the use of a random binary key to encrypt the text "Bathtub Pancake Ladybug".

```
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).
```

Example 2–1: Encryption using a random binary key

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.

Example 2–2 shows the use of a binary key to decrypt the text from the previous example.

```
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).
```

Example 2–2: Decryption using a random binary key
Example 2–3 shows the use of a password-based key to encrypt the text, "Matter Property Mass Solid". The password is "Migratory Blueberries".

```plaintext
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-HASH-ALGORITHM = "MD5".
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).
```

**Example 2–3: Encryption using a password-based key**

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 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).
Example 2–4 shows the use of a password-based key to decrypt the text from the previous example. The password is "Migratory Blueberries".

```abl
/* 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-HASH-ALGORITHM = "MD5".
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).
```

Example 2–4: Decryption using a password-based key

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 the “Generating encryption keys” section on page 2–54). If you store the secret key securely, like any cryptographic key, this results in a secure message digest.
Using cryptography to secure data

Generating message digests in ABL

ABL provides two built-in functions to generate message digests using the following syntax:

Syntax

| MD5-DIGEST | SHA1-DIGEST (data-to-digest [, secret-key-bytes]) |

The difference between these functions is the algorithm used to generate the digest. MD5-DIGEST uses the RSA Message Digest Hash Algorithm (MD5) and SHA1-DIGEST uses the United States Government Secure Hash Algorithm (SHA-1).

Both the data-to-digest and secret-key-bytes values accept CHARACTER, LONGCHAR, MEMPTR, or RAW values. If you use a CHARACTER or LONGCHAR, the function automatically converts the value to the UTF-8 code page before generating the digest, ensuring that the digest is code-page independent.

The result of each function is a RAW digest value that is 16 bytes long for an MD5 message digest and 20 bytes long for an SHA-1 message digest.

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 across 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 the “Managing and transporting crypto data” section on page 2–61.

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. 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**

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:

\[
ems = (((bms / 8) + 1) \times 8)
\]

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 \texttt{DECRYPT} function

This increase in size from binary to Base64-encoding is generally 33%. So, encrypted data in a \texttt{RAW} or \texttt{MEMPTR} variable becomes 33% larger after you convert to a Base64-encoded \texttt{CHARACTER} or \texttt{LONGCHAR} variable.
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 *OpenEdge Getting Started: Core Business Services*. This chapter provides basic information on how to perform the following tasks in ABL:

- 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
- Asserting the auditing identity
- Verifying that auditing is enabled
- Configuring auditing privileges
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 on authentication systems and trusted domain registries, see the information on identity management in OpenEdge Getting Started: Core Business Services.

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 the “Asserting the auditing identity” section on page 3–4. 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 Chapter 2, “Application Security.”

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 Chapter 2, “Application Security.”

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
- Using the OpenEdge session ID
- Application design considerations

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 *OpenEdge Deployment: Managing ABL Applications* and Chapter 2, “Application Security.” For more information on auditing privileges, see the “Configuring auditing privileges” section on page 3–7.

### 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 the “Managing audit event context” section on page 3–11.

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 *OpenEdge Deployment: Managing ABL Applications* and Chapter 2, “Application Security,” in this manual. For more information on auditing privileges, see the “Configuring auditing privileges” section on page 3–7.
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 UserID (-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 the “Managing audit event context” section on page 3–11.

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 Chapter 2, “Application Security.”

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 an n-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 AppServer. For more information, see Chapter 2, “Application Security.”
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 OpenEdge Getting Started: Core Business Services.

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. Table 3–1 lists the auditing privileges and the application functionality that they support.

Table 3–1: Auditing privilege capabilities

<table>
<thead>
<tr>
<th>This auditing privilege allows . . .</th>
<th>This set of capabilities . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application Audit Event Inserter (Optional)</td>
<td>Generating application events based on active audit policies</td>
</tr>
<tr>
<td>Audit Administrator</td>
<td>Configuring audit policies and reading audit data</td>
</tr>
<tr>
<td>Audit Data Archiver</td>
<td>Creating, reading, and deleting audit data</td>
</tr>
<tr>
<td>Audit Data Reporter</td>
<td>Reading audit policies and data</td>
</tr>
</tbody>
</table>

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 OpenEdge Getting Started: Core Business Services. For more information on setting audit privileges for users, see the Data Administration online help and OpenEdge Development: Basic Database Tools.

The following sections describe the essential ABL elements and requirements for writing these audit-aware applications and auditing tools.
Recording application events

OpenEdge provides core support for different types of auditing events at three basic levels of auditing:

- **Raw database record** — *Raw database 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 *OpenEdge Getting Started: Core Business Services*. 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
- ABL for generating application events
- Instructing OpenEdge to record an application event
- Managing audit event context
- Application design considerations

**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 OpenEdge Getting Started: Core Business Services.

**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. Table 3–2 lists the `AUDIT-CONTROL` handle methods that you can use to explicitly generate events in your application:

**Table 3–2: Methods for generating application events**

<table>
<thead>
<tr>
<th>This ABDIT-CONTROL method...</th>
<th>Provides this function...</th>
</tr>
</thead>
<tbody>
<tr>
<td>BEGIN-EVENT-GROUP( )</td>
<td>Starts an event group</td>
</tr>
<tr>
<td>CLEAR-APPL-CONTEXT( )</td>
<td>Clears the current application context</td>
</tr>
<tr>
<td>END-EVENT-GROUP( )</td>
<td>Ends the current event group</td>
</tr>
<tr>
<td>LOG-AUDIT-EVENT( )</td>
<td>Records an application-defined audit event</td>
</tr>
<tr>
<td>SET-APPL-CONTEXT( )</td>
<td>Sets the current application context</td>
</tr>
</tbody>
</table>

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 Chapter 2, “Application Security.”

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 OpenEdge Getting Started: Core Business Services. 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 the “Setting up application security for auditing” section on page 3–2.
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 the “Setting up application security for auditing” section on page 3–2.

- 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 *OpenEdge Development: Basic 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. Table 3–3 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 3–3: ABL elements for auditing client login session context

<table>
<thead>
<tr>
<th>ABL element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SESSION-ID</td>
<td>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 <code>_client-session</code> record created for the login session.</td>
</tr>
<tr>
<td>AUDIT-EVENT-CONTEXT</td>
<td>Client-principal object attribute that allows you to provide additional information to the audit event record's <code>_Event-context</code> 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.</td>
</tr>
<tr>
<td>AUTHENTICATION-FAILED( )</td>
<td>Client-principal object method that generates an audit event for a failed user login.</td>
</tr>
<tr>
<td>SEAL( )</td>
<td>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.</td>
</tr>
<tr>
<td>SET-CLIENT( )</td>
<td>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.</td>
</tr>
<tr>
<td>SET-DB-CLIENT</td>
<td>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.</td>
</tr>
<tr>
<td>LOGOUT( )</td>
<td>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 &quot;LOGOUT&quot; so that it cannot be validated to set an OpenEdge session or database connection ID.</td>
</tr>
</tbody>
</table>
If the option to record authenticated client sessions is set for an audit-enabled database, calling the \texttt{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 \texttt{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 Table 3–3 to maintain client login sessions, see Chapter 2, “Application Security.”

\section*{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 \texttt{SET-APPL-CONTEXT()} method on the \texttt{AUDIT-CONTROL} system handle. When you want to end the scope of the current application context, call the \texttt{CLEAR-APPL-CONTEXT()} method. The syntax for these methods follows:

\subsection*{Syntax}

\begin{verbatim}
\texttt{ctx-id = AUDIT-CONTROL:SET-APPL-CONTEXT}
\hspace{1cm} (\texttt{audit-event-ctx \[, event-detail \[, user-data \]}])
\end{verbatim}

\begin{verbatim}
\texttt{logical = AUDIT-CONTROL:CLEAR-APPL-CONTEXT( )}
\end{verbatim}

The $ctx\text{-}id$ 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 \texttt{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 \texttt{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:

```abl
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**

```abl
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).
...
cx-id = AUDIT-CONTROL:SET-APPL-CONTEXT
       ("Payroll app", "Federal tax calculation", user-task).
...
AUDIT-CONTROL:LOG-AUDIT-EVENT (34122, ctx-data).
...
cx-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 straightforward. 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 an n-tier application, especially an OERA-conformant application, where the AppServer 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 AppServer 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 AppServer 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\n```

For more information on using the Audit Policy Maintenance APIs and the audit policy tables, see the "Custom audit configuration tools" section on page 3–20. For more information on the Audit Policy Maintenance APIs themselves, see Appendix B, "Audit Policy Maintenance APIs."

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 *OpenEdge Getting Started: Core Business Services*.

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
- Custom audit archiving tools

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 *OpenEdge Getting Started: Core Business Services*. 
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 *OpenEdge Getting Started: Core Business Services* 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 *OpenEdge Getting Started: Core Business Services*.

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 AppServer.

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 Appendix B, “Audit Policy Maintenance APIs.”
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 OpenEdge Getting Started: Core Business Services.

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 the “Audit archiving security” section on page 3–24.
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 the “Generating client login session context” section on page 3–11.

- **_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 the “Audit archiving security” section on page 3–24.

For more information on these tables, see the sections on the audit data tables in *OpenEdge Getting Started: Core Business Services*.

### 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 the “Audit archiving security” section on page 3–24.
• 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 Chapter 2, “Application Security.”

• 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-ENCODEx and BASE64-DECODEx ABL functions. You can also convert between binary byte strings and strings of hexadecimal character pairs using the HEX-ENCODEx and HEX-DECODEx ABL functions. For more information, see the sections on managing cryptographic data in Chapter 2, “Application Security.”

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.
Part II

Input/Output Processes

Chapter 4, Handling User Input
Chapter 5, Alternate I/O Sources
Chapter 6, Colors and Fonts
Chapter 7, Creating Reports
This chapter explains how you can monitor user input from the keyboard or mouse. Chapter 5, “Alternate I/O Sources,” explains how to write procedures that accept input from a file.

This chapter covers 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 Figure 4–1, the F2 key label maps to key code 302. On most systems, this key code also maps to the ABL GO key function.

<table>
<thead>
<tr>
<th>Key label</th>
<th>Key code</th>
<th>Key function</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2</td>
<td>302</td>
<td>GO</td>
</tr>
</tbody>
</table>

Figure 4–1: 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 the “Using mouse buttons and events” section on page 4–22.

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 chapter on colors and fonts in this book and the chapter on user interface environments in OpenEdge Deployment: Managing 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 the “Monitoring keystrokes during data entry” section on page 4–28.
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. Table 4–1 shows the general order of precedence for ABL key code events.

### Table 4–1: ABL key input precedence

<table>
<thead>
<tr>
<th>Precedence (high to low)</th>
<th>Key code event</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>In the accelerator list for the current window</td>
</tr>
<tr>
<td>7</td>
<td>In the event list for an active user interface trigger</td>
</tr>
<tr>
<td>6</td>
<td>An active popup menu key</td>
</tr>
<tr>
<td>5</td>
<td>An active menu mnemonic key</td>
</tr>
<tr>
<td>4</td>
<td>An active field-level (button) mnemonic key (Windows only)</td>
</tr>
<tr>
<td>3</td>
<td>A special ABL internal key (built-in key function, like <strong>GO</strong></td>
</tr>
<tr>
<td>2</td>
<td>A key associated with a key function in the <code>PROTERMCP</code> file (UNIX only)</td>
</tr>
<tr>
<td>1</td>
<td>A standard input key (“A”, “B”, and so on)</td>
</tr>
</tbody>
</table>

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. Figure 4–2 shows how the conversions work, and which functions to use for each conversion.

![Figure 4–2: ABL key translation functions](image-url)
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:

```plaintext
KEYCODE("del")
```

The following function returns the value CTRL+G:

```plaintext
KEYLABEL(7).
```

See *OpenEdge Development: 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*. Figure 4–3 shows how ABL generates key codes when different keys are pressed.

<table>
<thead>
<tr>
<th>Key pressed</th>
<th>(Decimal)</th>
<th>(Binary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>F1</td>
<td>301</td>
<td>100101101</td>
</tr>
<tr>
<td>SHIFT-F1</td>
<td>813</td>
<td>100101101</td>
</tr>
<tr>
<td>ALT-F1</td>
<td>1325</td>
<td>100101101</td>
</tr>
<tr>
<td>CTRL-F1</td>
<td>2349</td>
<td>100101101</td>
</tr>
<tr>
<td>SHIFT-ALT-F1</td>
<td>1837</td>
<td>100101101</td>
</tr>
<tr>
<td>CTRL-ALT-F1</td>
<td>3373</td>
<td>100101101</td>
</tr>
<tr>
<td>CTRL-SHIFT-ALT-F1</td>
<td>3385</td>
<td>100101101</td>
</tr>
</tbody>
</table>

**Figure 4–3: 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 Table 4–2 are the preferred labels, but you can also use the labels in Table 4–3.

Table 4–2: Preferred key codes and key labels

<table>
<thead>
<tr>
<th>Key code</th>
<th>Key label</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>CTRL+@</td>
</tr>
<tr>
<td>1 through 7</td>
<td>CTRL+A through CTRL+G</td>
</tr>
<tr>
<td>8</td>
<td>BACKSPACE</td>
</tr>
<tr>
<td>9</td>
<td>TAB</td>
</tr>
<tr>
<td>10 through 12</td>
<td>CTRL+J through CTRL+L</td>
</tr>
<tr>
<td>13</td>
<td>ENTER (Windows)</td>
</tr>
<tr>
<td></td>
<td>RETURN (UNIX)</td>
</tr>
<tr>
<td>14 through 26</td>
<td>CTRL+N through CTRL+Z</td>
</tr>
<tr>
<td>27</td>
<td>ESC</td>
</tr>
<tr>
<td>28</td>
<td>CTRL+\</td>
</tr>
<tr>
<td>29</td>
<td>CTRL+]</td>
</tr>
<tr>
<td>30</td>
<td>CTRL+^</td>
</tr>
<tr>
<td>31</td>
<td>CTRL+–</td>
</tr>
<tr>
<td>32 through 126</td>
<td>Corresponding character</td>
</tr>
<tr>
<td>127</td>
<td>DEL</td>
</tr>
<tr>
<td>128 through 255</td>
<td>Corresponding extended character</td>
</tr>
<tr>
<td>300 through 399</td>
<td>F0 through F99</td>
</tr>
<tr>
<td>400 through 499</td>
<td>PF0 through PF99</td>
</tr>
<tr>
<td>501</td>
<td>CURSOR–UP</td>
</tr>
<tr>
<td>502</td>
<td>CURSOR–DOWN</td>
</tr>
<tr>
<td>503</td>
<td>CURSOR–RIGHT</td>
</tr>
<tr>
<td>504</td>
<td>CURSOR–LEFT</td>
</tr>
<tr>
<td>505</td>
<td>HOME</td>
</tr>
<tr>
<td>506</td>
<td>END</td>
</tr>
<tr>
<td>Key code</td>
<td>Key label</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------------</td>
</tr>
<tr>
<td>507</td>
<td>PAGE–UP</td>
</tr>
<tr>
<td>508</td>
<td>PAGE–DOWN</td>
</tr>
<tr>
<td>509</td>
<td>BACK–TAB</td>
</tr>
<tr>
<td>510</td>
<td>INS</td>
</tr>
<tr>
<td>511</td>
<td>HELP–KEY</td>
</tr>
<tr>
<td>512</td>
<td>DELETE</td>
</tr>
<tr>
<td>513</td>
<td>EXECUTE</td>
</tr>
<tr>
<td>514</td>
<td>PAGE</td>
</tr>
<tr>
<td>515</td>
<td>FIND</td>
</tr>
<tr>
<td>516</td>
<td>INS–LINE</td>
</tr>
<tr>
<td>517</td>
<td>DEL–LINE</td>
</tr>
<tr>
<td>518</td>
<td>LINE–ERASE</td>
</tr>
<tr>
<td>519</td>
<td>PAGE–ERASE</td>
</tr>
<tr>
<td>520</td>
<td>SHIFT+BACKSPACE</td>
</tr>
<tr>
<td>521</td>
<td>SHIFT+TAB</td>
</tr>
<tr>
<td>522</td>
<td>EXIT</td>
</tr>
<tr>
<td>535</td>
<td>ERASE</td>
</tr>
<tr>
<td>536</td>
<td>WHITE</td>
</tr>
<tr>
<td>537</td>
<td>BLUE</td>
</tr>
<tr>
<td>538</td>
<td>RED</td>
</tr>
<tr>
<td>539</td>
<td>RESET</td>
</tr>
<tr>
<td>541</td>
<td>CTRL+BREAK</td>
</tr>
<tr>
<td>551 through 560</td>
<td>U1 through U10</td>
</tr>
<tr>
<td>609</td>
<td>MOUSE–SELECT–UP</td>
</tr>
<tr>
<td>610</td>
<td>MOUSE–MOVE–UP</td>
</tr>
<tr>
<td>611</td>
<td>MOUSE–MENU–UP</td>
</tr>
<tr>
<td>612</td>
<td>MOUSE–EXTEND–UP</td>
</tr>
<tr>
<td>617</td>
<td>MOUSE–SELECT–DOWN</td>
</tr>
<tr>
<td>618</td>
<td>MOUSE–MOVE–DOWN</td>
</tr>
</tbody>
</table>
Table 4–2: Preferred key codes and key labels

<table>
<thead>
<tr>
<th>Key code</th>
<th>Key label</th>
</tr>
</thead>
<tbody>
<tr>
<td>619</td>
<td>MOUSE–MENU–DOWN</td>
</tr>
<tr>
<td>620</td>
<td>MOUSE–EXTEND–DOWN</td>
</tr>
<tr>
<td>625</td>
<td>MOUSE–SELECT–CLICK</td>
</tr>
<tr>
<td>626</td>
<td>MOUSE–MOVE–CLICK</td>
</tr>
<tr>
<td>627</td>
<td>MOUSE–MENU–CLICK</td>
</tr>
<tr>
<td>628</td>
<td>MOUSE–EXTEND–CLICK</td>
</tr>
<tr>
<td>640</td>
<td>MOUSE–MOVE</td>
</tr>
<tr>
<td>649</td>
<td>MOUSE–SELECT–DBLCLICK</td>
</tr>
<tr>
<td>650</td>
<td>MOUSE–MOVE–DBLCLICK</td>
</tr>
<tr>
<td>651</td>
<td>MOUSE–MENU–DBLCLICK</td>
</tr>
<tr>
<td>652</td>
<td>MOUSE–EXTEND–DBLCLICK</td>
</tr>
<tr>
<td>812 through 911</td>
<td>SHIFT+F0 through SHIFT+F99</td>
</tr>
<tr>
<td>912 through 1011</td>
<td>SHIFT+PF0 through SHIFT+PF99</td>
</tr>
<tr>
<td>1013</td>
<td>SHIFT+CURSOR–UP</td>
</tr>
<tr>
<td>1014</td>
<td>SHIFT+CURSOR–DOWN</td>
</tr>
<tr>
<td>1015</td>
<td>SHIFT+CURSOR–RIGHT</td>
</tr>
<tr>
<td>1016</td>
<td>SHIFT+CURSOR–LEFT</td>
</tr>
<tr>
<td>1017</td>
<td>SHIFT+HOME</td>
</tr>
<tr>
<td>1018</td>
<td>SHIFT+END</td>
</tr>
<tr>
<td>1019</td>
<td>SHIFT+PAGE–UP</td>
</tr>
<tr>
<td>1020</td>
<td>SHIFT+PAGE–DOWN</td>
</tr>
<tr>
<td>1021</td>
<td>SHIFT+BACK–TAB</td>
</tr>
<tr>
<td>1022</td>
<td>SHIFT+INS</td>
</tr>
<tr>
<td>1023</td>
<td>SHIFT+HELP–KEY</td>
</tr>
<tr>
<td>1024</td>
<td>ALT+CTRL+@</td>
</tr>
<tr>
<td>1025 through 1031</td>
<td>ALT+CTRL+A through ALT+CTRL+G</td>
</tr>
<tr>
<td>1032</td>
<td>ALT+BACKSPACE</td>
</tr>
<tr>
<td>1033</td>
<td>ALT+TAB</td>
</tr>
<tr>
<td>1034 through 1036</td>
<td>ALT+CTRL+J through ALT+CTRL+L</td>
</tr>
<tr>
<td>Key code</td>
<td>Key label</td>
</tr>
<tr>
<td>---------</td>
<td>--------------------------------------------</td>
</tr>
<tr>
<td>1037</td>
<td>ALT+RETURN</td>
</tr>
<tr>
<td>1038 through 1050</td>
<td>ALT+CTRL+N through ALT+CTRL+Z</td>
</tr>
<tr>
<td>1051</td>
<td>ALT+ESC</td>
</tr>
<tr>
<td>1052</td>
<td>ALT+CTRL+\</td>
</tr>
<tr>
<td>1053</td>
<td>ALT+CTRL+]</td>
</tr>
<tr>
<td>1054</td>
<td>ALT+CTRL+^</td>
</tr>
<tr>
<td>1055</td>
<td>ALT+CTRL+_</td>
</tr>
<tr>
<td>1057 through 1150</td>
<td>ALT+ corresponding character</td>
</tr>
<tr>
<td>1151</td>
<td>ALT+DEL</td>
</tr>
<tr>
<td>1152 through 1279</td>
<td>ALT+ corresponding extended character</td>
</tr>
<tr>
<td>1324 through 1423</td>
<td>ALT+F0 through ALT+F99</td>
</tr>
<tr>
<td>1424 through 1523</td>
<td>ALT+PF0 through ALT+PF99</td>
</tr>
<tr>
<td>1525</td>
<td>ALT+CURSOR–UP</td>
</tr>
<tr>
<td>1526</td>
<td>ALT+CURSOR–DOWN</td>
</tr>
<tr>
<td>1527</td>
<td>ALT+CURSOR–RIGHT</td>
</tr>
<tr>
<td>1528</td>
<td>ALT+CURSOR–LEFT</td>
</tr>
<tr>
<td>1529</td>
<td>ALT+HOME</td>
</tr>
<tr>
<td>1530</td>
<td>ALT+END</td>
</tr>
<tr>
<td>1531</td>
<td>ALT+PAGE–UP</td>
</tr>
<tr>
<td>1532</td>
<td>ALT+PAGE–DOWN</td>
</tr>
<tr>
<td>1533</td>
<td>ALT+BACK–TAB</td>
</tr>
<tr>
<td>1534</td>
<td>ALT+INS</td>
</tr>
<tr>
<td>1535</td>
<td>ALT+HELP–KEY</td>
</tr>
<tr>
<td>1536</td>
<td>ALT+DELETE</td>
</tr>
<tr>
<td>1537</td>
<td>ALT+EXECUTE</td>
</tr>
<tr>
<td>1538</td>
<td>ALT+PAGE</td>
</tr>
<tr>
<td>1539</td>
<td>ALT+FIND</td>
</tr>
<tr>
<td>1540</td>
<td>ALT+INS–LINE</td>
</tr>
<tr>
<td>1541</td>
<td>ALT+DEL–LINE</td>
</tr>
<tr>
<td>Key code</td>
<td>Key label</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>1542</td>
<td>ALT+LINE–ERASE</td>
</tr>
<tr>
<td>1543</td>
<td>ALT+PAGE–ERASE</td>
</tr>
<tr>
<td>1544</td>
<td>ALT+SHIFT+BACKSPACE</td>
</tr>
<tr>
<td>1545</td>
<td>ALT+SHIFT+TAB</td>
</tr>
<tr>
<td>1546</td>
<td>ALT+EXIT</td>
</tr>
<tr>
<td>1549 through 1558</td>
<td>ALT+U1 through ALT+U10</td>
</tr>
<tr>
<td>1559</td>
<td>ALT+ERASE</td>
</tr>
<tr>
<td>1560</td>
<td>ALT+WHITE</td>
</tr>
<tr>
<td>1561</td>
<td>ALT+BLUE</td>
</tr>
<tr>
<td>1562</td>
<td>ALT+RED</td>
</tr>
<tr>
<td>1563</td>
<td>ALT+RESET</td>
</tr>
<tr>
<td>1565</td>
<td>ALT+ctrl+BREAK</td>
</tr>
<tr>
<td>1575 through 1584</td>
<td>ALT+U1 through ALT+U10</td>
</tr>
<tr>
<td>1663</td>
<td>SHIFT+ALT+DEL</td>
</tr>
<tr>
<td>1664 through 1791</td>
<td>SHIFT+ALT+ corresponding extended character</td>
</tr>
<tr>
<td>1836 through 1935</td>
<td>SHIFT+ALT+F0 through SHIFT+ALT+F99</td>
</tr>
<tr>
<td>1936 through 2035</td>
<td>SHIFT+ALT+PF0 through SHIFT+ALT+PF99</td>
</tr>
<tr>
<td>2037</td>
<td>SHIFT+ALT+CURSOR–UP</td>
</tr>
<tr>
<td>2038</td>
<td>SHIFT+ALT+CURSOR–DOWN</td>
</tr>
<tr>
<td>2039</td>
<td>SHIFT+ALT+CURSOR–RIGHT</td>
</tr>
<tr>
<td>2040</td>
<td>SHIFT+ALT+CURSOR–LEFT</td>
</tr>
<tr>
<td>2041</td>
<td>SHIFT+ALT+HOME</td>
</tr>
<tr>
<td>2042</td>
<td>SHIFT+ALT+END</td>
</tr>
<tr>
<td>2043</td>
<td>SHIFT+ALT+PAGE–UP</td>
</tr>
<tr>
<td>2044</td>
<td>SHIFT+ALT+PAGE–DOWN</td>
</tr>
<tr>
<td>2045</td>
<td>SHIFT+ALT+BACK–TAB</td>
</tr>
<tr>
<td>2046</td>
<td>SHIFT+ALT+INS</td>
</tr>
<tr>
<td>2047</td>
<td>SHIFT+ALT+HELP–KEY</td>
</tr>
<tr>
<td>2056</td>
<td>CTRL+ BACKSPACE</td>
</tr>
<tr>
<td>Key code</td>
<td>Key label</td>
</tr>
<tr>
<td>---------</td>
<td>-----------------------------------------------</td>
</tr>
<tr>
<td>2057</td>
<td>CTRL+TAB</td>
</tr>
<tr>
<td>2075</td>
<td>CTRL+ESC</td>
</tr>
<tr>
<td>2080 through 2111</td>
<td>CTRL+ corresponding character</td>
</tr>
<tr>
<td>2171 through 2174</td>
<td>CTRL+ corresponding character</td>
</tr>
<tr>
<td>2175</td>
<td>CTRL+DEL</td>
</tr>
<tr>
<td>2209 through 2303</td>
<td>CTRL+ corresponding extended character</td>
</tr>
<tr>
<td>2348 through 2447</td>
<td>CTRL+F0 through CTRL+F99</td>
</tr>
<tr>
<td>2448 through 2547</td>
<td>CTRL+PF0 through CTRL+PF99</td>
</tr>
<tr>
<td>2549</td>
<td>CTRL+CURSOR–UP</td>
</tr>
<tr>
<td>2550</td>
<td>CTRL+CURSOR–DOWN</td>
</tr>
<tr>
<td>2551</td>
<td>CTRL+CURSOR–RIGHT</td>
</tr>
<tr>
<td>2552</td>
<td>CTRL+CURSOR–LEFT</td>
</tr>
<tr>
<td>2553</td>
<td>CTRL+HOME</td>
</tr>
<tr>
<td>2554</td>
<td>CTRL+END</td>
</tr>
<tr>
<td>2555</td>
<td>CTRL+PAGE–UP</td>
</tr>
<tr>
<td>2556</td>
<td>CTRL+PAGE–DOWN</td>
</tr>
<tr>
<td>2557</td>
<td>CTRL+BACK–TAB</td>
</tr>
<tr>
<td>2558</td>
<td>CTRL+INS</td>
</tr>
<tr>
<td>2559</td>
<td>CTRL+HELP–KEY</td>
</tr>
<tr>
<td>2560</td>
<td>CTRL+DELETE</td>
</tr>
<tr>
<td>2561</td>
<td>CTRL+EXECUTE</td>
</tr>
<tr>
<td>2562</td>
<td>CTRL+PAGE</td>
</tr>
<tr>
<td>2563</td>
<td>CTRL+FIND</td>
</tr>
<tr>
<td>2564</td>
<td>CTRL+INS–LINE</td>
</tr>
<tr>
<td>2565</td>
<td>CTRL+DEL–LINE</td>
</tr>
<tr>
<td>2566</td>
<td>CTRL+LINE–ERASE</td>
</tr>
<tr>
<td>2567</td>
<td>CTRL+PAGE–ERASE</td>
</tr>
<tr>
<td>2568</td>
<td>CTRL+SHIFT+BACKSPACE</td>
</tr>
<tr>
<td>2569</td>
<td>CTRL+SHIFT+TAB</td>
</tr>
<tr>
<td>Key code</td>
<td>Key label</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------------------</td>
</tr>
<tr>
<td>2570</td>
<td>CTRL+EXIT</td>
</tr>
<tr>
<td>2573</td>
<td>CTRL+SHIFT+RETURN</td>
</tr>
<tr>
<td>2583</td>
<td>CTRL+ERASE</td>
</tr>
<tr>
<td>2584</td>
<td>CTRL+WHITE</td>
</tr>
<tr>
<td>2585</td>
<td>CTRL+BLUE</td>
</tr>
<tr>
<td>2586</td>
<td>CTRL+RED</td>
</tr>
<tr>
<td>2587</td>
<td>CTRL+RESET</td>
</tr>
<tr>
<td>2559 through 2608</td>
<td>CTRL+U1 through CTRL+U10</td>
</tr>
<tr>
<td>2860 through 2859</td>
<td>CTRL+SHIFT+F0 through CTRL+SHIFT+F99</td>
</tr>
<tr>
<td>2960 through 3059</td>
<td>CTRL+SHIFT+PF0 through CTRL+SHIFT+PF99</td>
</tr>
<tr>
<td>3061</td>
<td>CTRL+SHIFT+CURSOR–UP</td>
</tr>
<tr>
<td>3062</td>
<td>CTRL+SHIFT+CURSOR–DOWN</td>
</tr>
<tr>
<td>3063</td>
<td>CTRL+SHIFT+CURSOR–RIGHT</td>
</tr>
<tr>
<td>3064</td>
<td>CTRL+SHIFT+CURSOR–LEFT</td>
</tr>
<tr>
<td>3065</td>
<td>CTRL+SHIFT+HOME</td>
</tr>
<tr>
<td>3066</td>
<td>CTRL+SHIFT+END</td>
</tr>
<tr>
<td>3067</td>
<td>CTRL+SHIFT+PAGE–UP</td>
</tr>
<tr>
<td>3068</td>
<td>CTRL+SHIFT+PAGE–DOWN</td>
</tr>
<tr>
<td>3069</td>
<td>CTRL+SHIFT+BACK–TAB</td>
</tr>
<tr>
<td>3070</td>
<td>CTRL+SHIFT+INS</td>
</tr>
<tr>
<td>3071</td>
<td>CTRL+SHIFT+HELP–KEY</td>
</tr>
<tr>
<td>3104 through 3135</td>
<td>CTRL+ALT+ corresponding character</td>
</tr>
<tr>
<td>3163</td>
<td>CTRL+ALT+{</td>
</tr>
<tr>
<td>3164</td>
<td>CTRL+ALT+\</td>
</tr>
<tr>
<td>3165</td>
<td>CTRL+ALT+}</td>
</tr>
<tr>
<td>3166</td>
<td>CTRL+ALT+^{</td>
</tr>
<tr>
<td>3167</td>
<td>CTRL+ALT+_</td>
</tr>
<tr>
<td>3168</td>
<td>CTRL+ALT+^</td>
</tr>
<tr>
<td>3195 through 3198</td>
<td>CTRL+ALT+ corresponding character</td>
</tr>
</tbody>
</table>
### Table 4–2: Preferred key codes and key labels

<table>
<thead>
<tr>
<th>Key code</th>
<th>Key label</th>
</tr>
</thead>
<tbody>
<tr>
<td>3199</td>
<td>CTRL+ALT+DEL</td>
</tr>
<tr>
<td>3233 through 3327</td>
<td>CTRL+ALT+ corresponding extended character</td>
</tr>
<tr>
<td>3372 through 3471</td>
<td>CTRL+ALT+F0 through CTRL+ALT+F10</td>
</tr>
<tr>
<td>3472 through 3571</td>
<td>CTRL+ALT+PF0 through CTRL+ALT+PF99</td>
</tr>
<tr>
<td>3573</td>
<td>CTRL+ALT+CURSOR–UP</td>
</tr>
<tr>
<td>3574</td>
<td>CTRL+ALT+CURSOR–DOWN</td>
</tr>
<tr>
<td>3575</td>
<td>CTRL+ALT+CURSOR–RIGHT</td>
</tr>
<tr>
<td>3576</td>
<td>CTRL+ALT+CURSOR–LEFT</td>
</tr>
<tr>
<td>3577</td>
<td>CTRL+ALT+HOME</td>
</tr>
<tr>
<td>3578</td>
<td>CTRL+ALT+END</td>
</tr>
<tr>
<td>3579</td>
<td>CTRL+ALT+PAGE–UP</td>
</tr>
<tr>
<td>3580</td>
<td>CTRL+ALT+PAGE–DOWN</td>
</tr>
<tr>
<td>3581</td>
<td>CTRL+ALT+BACK–TAB</td>
</tr>
<tr>
<td>3582</td>
<td>CTRL+ALT+INS</td>
</tr>
<tr>
<td>3583</td>
<td>CTRL+ALT+HELP–KEY</td>
</tr>
<tr>
<td>3585</td>
<td>CTRL+ALT+EXECUTE</td>
</tr>
<tr>
<td>3586</td>
<td>CTRL+ALT+PAGE</td>
</tr>
<tr>
<td>3587</td>
<td>CTRL+ALT+FIND</td>
</tr>
<tr>
<td>3588</td>
<td>CTRL+ALT+INS–LINE</td>
</tr>
<tr>
<td>3589</td>
<td>CTRL+ALT+DEL–LINE</td>
</tr>
<tr>
<td>3590</td>
<td>CTRL+ALT+LINE–ERASE</td>
</tr>
<tr>
<td>3591</td>
<td>CTRL+ALT+PAGE–ERASE</td>
</tr>
<tr>
<td>3592</td>
<td>CTRL+SHIFT+ALT+BACKSPACE</td>
</tr>
<tr>
<td>3593</td>
<td>CTRL+SHIFT+ALT+TAB</td>
</tr>
<tr>
<td>3594</td>
<td>CTRL+ALT+EXIT</td>
</tr>
<tr>
<td>3597</td>
<td>CTRL+SHIFT+ALT+RETURN</td>
</tr>
<tr>
<td>3607</td>
<td>CTRL+ALT+ERASE</td>
</tr>
<tr>
<td>3608</td>
<td>CTRL+ALT+WHITE</td>
</tr>
<tr>
<td>3609</td>
<td>CTRL+ALT+BLUE</td>
</tr>
<tr>
<td>Key code</td>
<td>Key label</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------------------------------</td>
</tr>
<tr>
<td>3610</td>
<td>CTRL+ALT+RED</td>
</tr>
<tr>
<td>3611</td>
<td>CTRL+ALT+RESET</td>
</tr>
<tr>
<td>3623 through 3632</td>
<td>CTRL+ALT+U1 through CTRL+ALT+U10</td>
</tr>
<tr>
<td>3884 through 3983</td>
<td>CTRL+SHIFT+ALT+F0 through CTRL+SHIFT+ALT+F99</td>
</tr>
<tr>
<td>3994 through 4083</td>
<td>CTRL+SHIFT+ALT+PF0 through CTRL+SHIFT+ALT+PF99</td>
</tr>
<tr>
<td>4085</td>
<td>CTRL+SHIFT+ALT+CURSOR–UP</td>
</tr>
<tr>
<td>4086</td>
<td>CTRL+SHIFT+ALT+CURSOR–DOWN</td>
</tr>
<tr>
<td>4087</td>
<td>CTRL+SHIFT+ALT+CURSOR–RIGHT</td>
</tr>
<tr>
<td>4088</td>
<td>CTRL+SHIFT+ALT+CURSOR–LEFT</td>
</tr>
<tr>
<td>4089</td>
<td>CTRL+SHIFT+ALT+HOME</td>
</tr>
<tr>
<td>4090</td>
<td>CTRL+SHIFT+ALT+END</td>
</tr>
<tr>
<td>4091</td>
<td>CTRL+SHIFT+ALT+PAGE–UP</td>
</tr>
<tr>
<td>4092</td>
<td>CTRL+SHIFT+ALT+PAGE–DOWN</td>
</tr>
<tr>
<td>4093</td>
<td>CTRL+SHIFT+ALT+BACK–TAB</td>
</tr>
<tr>
<td>4094</td>
<td>CTRL+SHIFT+ALT+INS</td>
</tr>
<tr>
<td>4095</td>
<td>CTRL+SHIFT+ALT+HELP–KEY</td>
</tr>
</tbody>
</table>
Table 4–2 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. Table 4–3 lists the alternate key labels.

**Table 4–3: Alternate key codes and key labels**

<table>
<thead>
<tr>
<th>Key code</th>
<th>Alternate key label</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>BELL</td>
</tr>
<tr>
<td>8</td>
<td>BS</td>
</tr>
<tr>
<td>10</td>
<td>LINEFEED, LF</td>
</tr>
<tr>
<td>12</td>
<td>FORMFEED, FF</td>
</tr>
<tr>
<td>13</td>
<td>RETURN (Windows)</td>
</tr>
<tr>
<td></td>
<td>ENTER (UNIX)</td>
</tr>
<tr>
<td>27</td>
<td>ESCAPE</td>
</tr>
<tr>
<td>127</td>
<td>CANCEL</td>
</tr>
<tr>
<td>501</td>
<td>UP, UP→ARROW</td>
</tr>
<tr>
<td>502</td>
<td>DOWN, DOWN→ARROW</td>
</tr>
<tr>
<td>503</td>
<td>RIGHT, RIGHT→ARROW</td>
</tr>
<tr>
<td>504</td>
<td>LEFT, LEFT→ARROW</td>
</tr>
<tr>
<td>505</td>
<td>ESC H</td>
</tr>
<tr>
<td>507</td>
<td>PGUP, PREV→PAGE, PREV→SCRN</td>
</tr>
<tr>
<td>508</td>
<td>PGDN, NEXT→PAGE, NEXT→SCRN</td>
</tr>
<tr>
<td>509</td>
<td>SHIFT+TAB (UNIX)</td>
</tr>
<tr>
<td></td>
<td>BACK→TAB (Windows)</td>
</tr>
<tr>
<td>510</td>
<td>INSERT, INS→CHAR, INS→C, INSERT→HERE</td>
</tr>
<tr>
<td>512</td>
<td>DEL→CHAR, DELETE→CHAR, DEL→C</td>
</tr>
<tr>
<td>516</td>
<td>INS→L, LINE→INS</td>
</tr>
<tr>
<td>517</td>
<td>DEL→L, LINE→DEL</td>
</tr>
</tbody>
</table>
Table 4–4 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. Table 4–4 lists possible mappings.

<table>
<thead>
<tr>
<th>Key function</th>
<th>Windows graphical interface</th>
<th>UNIX character interface</th>
<th>Windows character interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABORT</td>
<td></td>
<td>CTRL+\</td>
<td></td>
</tr>
<tr>
<td>APPEND–LINE</td>
<td></td>
<td>CTRL+A</td>
<td>CTRL+A</td>
</tr>
<tr>
<td>BACK–TAB</td>
<td>SHIFT+TAB</td>
<td>CTRL+U</td>
<td>CTRL+U SHIFT+TAB</td>
</tr>
<tr>
<td>BACKSPACE</td>
<td>BACKSPACE</td>
<td>BACKSPACE CTRL+H DEL=CHAR</td>
<td>BACKSPACE CTRL+H</td>
</tr>
<tr>
<td>BELL</td>
<td>CTRL+G BELL</td>
<td>BELL</td>
<td></td>
</tr>
<tr>
<td>BLOCK</td>
<td></td>
<td>CTRL+V</td>
<td>CTRL+V</td>
</tr>
<tr>
<td>BOTTOM–COLUMN</td>
<td></td>
<td>ESC CTRL+B</td>
<td></td>
</tr>
<tr>
<td>BREAK–LINE</td>
<td></td>
<td>ESC B</td>
<td></td>
</tr>
<tr>
<td>CANCEL–PICK</td>
<td></td>
<td>ESC CTRL+X</td>
<td></td>
</tr>
<tr>
<td>CHOICES</td>
<td></td>
<td>ESC BACKSPACE ESC CTRL+H</td>
<td></td>
</tr>
<tr>
<td>CLEAR</td>
<td></td>
<td>F8 CTRL+Z</td>
<td>F8 CTRL+Z</td>
</tr>
<tr>
<td>CLOSE</td>
<td>F8</td>
<td>ESC Z</td>
<td>CTRL+ALT+Z F8</td>
</tr>
<tr>
<td>COMPILE</td>
<td>SHIFT+F2</td>
<td>ESC P</td>
<td>CTRL+ALT+P</td>
</tr>
<tr>
<td>COPY</td>
<td></td>
<td>F11 ESC C</td>
<td>F11 CTRL+ALT+C</td>
</tr>
<tr>
<td>CURSOR–DOWN</td>
<td>CURSOR–DOWN</td>
<td>CURSOR–DOWN CTRL+J</td>
<td>CURSOR–DOWN CTRL+J</td>
</tr>
<tr>
<td>CURSOR–LEFT</td>
<td>CURSOR–LEFT</td>
<td>CURSOR–LEFT CTRL+O</td>
<td>CURSOR–LEFT CTRL+O</td>
</tr>
<tr>
<td>CURSOR–RIGHT</td>
<td>CURSOR–RIGHT</td>
<td>CURSOR–RIGHT CTRL+L</td>
<td>CURSOR–RIGHT CTRL+L</td>
</tr>
<tr>
<td>Key function</td>
<td>Windows graphical interface</td>
<td>UNIX character interface</td>
<td>Windows character interface</td>
</tr>
<tr>
<td>-----------------------</td>
<td>-----------------------------</td>
<td>--------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>CUT</td>
<td>CTRL+X</td>
<td>F10 ESC X</td>
<td>F10 CTRL+ALT+X</td>
</tr>
<tr>
<td>DEFAULT–POP–UP</td>
<td>SHIFT+F10</td>
<td>ESC U</td>
<td>SHIFT+F4 CTRL+ALT+U</td>
</tr>
<tr>
<td>DELETE–CHARACTER</td>
<td>DEL</td>
<td>DEL DELETE</td>
<td>DELETE</td>
</tr>
<tr>
<td>DELETE–COLUMN</td>
<td>-</td>
<td>ESC CTRL+Z</td>
<td>-</td>
</tr>
<tr>
<td>DELETE–END–LINE</td>
<td>-</td>
<td>ESC K</td>
<td>CTRL+ALT+K</td>
</tr>
<tr>
<td>DELETE–FIELD</td>
<td>-</td>
<td>ESC CTRL+D</td>
<td>-</td>
</tr>
<tr>
<td>DELETE–LINE</td>
<td>-</td>
<td>CTRL+D</td>
<td>CTRL+D</td>
</tr>
<tr>
<td>DELETE–WORD</td>
<td>-</td>
<td>ESC D</td>
<td>CTRL+ALT+D</td>
</tr>
<tr>
<td>EDITOR–BACKTAB</td>
<td>-</td>
<td>CTRL+B</td>
<td>CTRL+B</td>
</tr>
<tr>
<td>EDITOR–TAB</td>
<td>-</td>
<td>CTRL+G TAB</td>
<td>CTRL+G TAB</td>
</tr>
<tr>
<td>END</td>
<td>END</td>
<td>END</td>
<td>END</td>
</tr>
<tr>
<td>END–ERROR</td>
<td>ESC</td>
<td>F4 CTRL+E</td>
<td>F4 ESC CTRL+E</td>
</tr>
<tr>
<td>ENTER–MENUBAR</td>
<td>ALT</td>
<td>F3 PF3 ESC M</td>
<td>F3 ALT</td>
</tr>
<tr>
<td>EXIT</td>
<td>-</td>
<td>ESC Q</td>
<td>CTRL+ALT+Q</td>
</tr>
<tr>
<td>FIND</td>
<td>CTRL+F</td>
<td>CTRL+F</td>
<td>CTRL+F</td>
</tr>
<tr>
<td>FIND–NEXT</td>
<td>F9</td>
<td>ESC F</td>
<td>CTRL+ALT+F</td>
</tr>
<tr>
<td>FIND–PREVIOUS</td>
<td>SHIFT+F9</td>
<td>ESC I</td>
<td>CTRL+ALT+I</td>
</tr>
<tr>
<td>GET</td>
<td>F3</td>
<td>F5 ESC O</td>
<td>F5 CTRL+ALT+O</td>
</tr>
<tr>
<td>GO</td>
<td>F2</td>
<td>F1 CTRL+X</td>
<td>F1 CTRL+X</td>
</tr>
<tr>
<td>GOTO</td>
<td>CTRL+G</td>
<td>ESC G</td>
<td>CTRL+ALT+G</td>
</tr>
<tr>
<td>HELP</td>
<td>F1</td>
<td>ESC ?</td>
<td>_</td>
</tr>
<tr>
<td>HOME</td>
<td>HOME</td>
<td>ESC, ESC H</td>
<td>HOME</td>
</tr>
<tr>
<td>INSERT–COLUMN</td>
<td>-</td>
<td>ESC CTRL+N</td>
<td>-</td>
</tr>
<tr>
<td>INSERT–FIELD</td>
<td>-</td>
<td>ESC CTRL+G</td>
<td>-</td>
</tr>
</tbody>
</table>
Table 4–4: **ABL key functions**

<table>
<thead>
<tr>
<th>Key function</th>
<th>Windows graphical interface</th>
<th>UNIX character interface</th>
<th>Windows character interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSERT–FIELD–DATA</td>
<td>_</td>
<td>ESC CTRL+F</td>
<td>_</td>
</tr>
<tr>
<td>INSERT–FIELD–LABEL</td>
<td>_</td>
<td>ESC CTRL+E</td>
<td>_</td>
</tr>
<tr>
<td>INSERT–MODE</td>
<td>INSERT</td>
<td>F9 CTRL+T</td>
<td>INSERT F9 CTRL+T</td>
</tr>
<tr>
<td>LEFT–END</td>
<td>HOME</td>
<td>ESC CURSOR–LEFT</td>
<td>ALT+CURSOR–LEFT</td>
</tr>
<tr>
<td>MAIN–MENU</td>
<td>_</td>
<td>ESC RETURN ESC CTRL+M</td>
<td>_</td>
</tr>
<tr>
<td>MOVE</td>
<td>_</td>
<td>ESC CTRL+V</td>
<td>_</td>
</tr>
<tr>
<td>NEW</td>
<td>SHIFT+F3</td>
<td>ESC N</td>
<td>CTRL+ALT+N</td>
</tr>
<tr>
<td>NEW–LINE</td>
<td>_</td>
<td>CTRL+N</td>
<td>CTRL+N</td>
</tr>
<tr>
<td>NEXT–ERROR</td>
<td>_</td>
<td>ESC E</td>
<td>_</td>
</tr>
<tr>
<td>NEXT–FRAME</td>
<td>F6</td>
<td>ESC TAB ESC CTRL+I</td>
<td>_</td>
</tr>
<tr>
<td>NEXT–WORD</td>
<td>_</td>
<td>CTRL+W</td>
<td>_</td>
</tr>
<tr>
<td>OPEN–LINE–ABOVE</td>
<td>_</td>
<td>ESC L</td>
<td>CTRL+ALT+L</td>
</tr>
<tr>
<td>OPTIONS</td>
<td>_</td>
<td>ESC CTRL+O</td>
<td>_</td>
</tr>
<tr>
<td>PAGE–LEFT</td>
<td>_</td>
<td>ESC W</td>
<td>_</td>
</tr>
<tr>
<td>PAGE–RIGHT</td>
<td>_</td>
<td>ESC Y</td>
<td>_</td>
</tr>
<tr>
<td>PASTE</td>
<td>CTRL+V</td>
<td>F12 ESC V</td>
<td>F12 CTRL+ALT+V</td>
</tr>
<tr>
<td>PICK</td>
<td>_</td>
<td>ESC CTRL+P</td>
<td>_</td>
</tr>
<tr>
<td>PICK–AREA</td>
<td>_</td>
<td>ESC CTRL+W</td>
<td>_</td>
</tr>
<tr>
<td>PICK–BOTH</td>
<td>_</td>
<td>ESC CTRL+Q</td>
<td>_</td>
</tr>
<tr>
<td>PREV–FRAME</td>
<td>SHIFT+F6</td>
<td>ESC CTRL+U</td>
<td>CTRL+SHIFT+TAB</td>
</tr>
<tr>
<td>PREV–WORD</td>
<td>_</td>
<td>CTRL+P</td>
<td>CTRL+P</td>
</tr>
</tbody>
</table>
### Table 4-4: ABL key functions

<table>
<thead>
<tr>
<th>Key function</th>
<th>Windows graphical interface</th>
<th>UNIX character interface</th>
<th>Windows character interface</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUT</td>
<td>F6</td>
<td>F6</td>
<td>F6</td>
</tr>
<tr>
<td>RECALL</td>
<td></td>
<td>F7 CTRL+R</td>
<td></td>
</tr>
<tr>
<td>REPLACE</td>
<td></td>
<td>ESC R</td>
<td></td>
</tr>
<tr>
<td>REPORTS</td>
<td></td>
<td>ESC CTRL+A</td>
<td></td>
</tr>
<tr>
<td>RESUME–DISPLAY</td>
<td></td>
<td>CTRL+Q</td>
<td></td>
</tr>
<tr>
<td>RETURN</td>
<td>ENTER RETURN ENTER CTRL+M</td>
<td>RETURN ENTER CTRL+M</td>
<td>ENTER CTRL+M</td>
</tr>
<tr>
<td>RIGHT–END</td>
<td></td>
<td>ESC CURSOR–RIGHT</td>
<td>ALT+CURSOR–RIGHT</td>
</tr>
<tr>
<td>SAVE–AS</td>
<td>SHIFT+F6</td>
<td>ESC A</td>
<td>CTRL+ALT+A</td>
</tr>
<tr>
<td>SCROLL–LEFT</td>
<td></td>
<td>ESC CTRL+L</td>
<td></td>
</tr>
<tr>
<td>SCROLL–MODE</td>
<td></td>
<td>ESC T</td>
<td>CTRL+ALT+T</td>
</tr>
<tr>
<td>SCROLL–RIGHT</td>
<td></td>
<td>ESC CTRL+R</td>
<td></td>
</tr>
<tr>
<td>SETTINGS</td>
<td></td>
<td>ESC CTRL+@</td>
<td></td>
</tr>
<tr>
<td>STOP</td>
<td>CTRL+BREAK</td>
<td>CTRL+C</td>
<td>CTRL+C</td>
</tr>
<tr>
<td>STOP–DISPLAY</td>
<td></td>
<td>CTRL+S</td>
<td></td>
</tr>
<tr>
<td>TAB</td>
<td>TAB</td>
<td>TAB CTRL+I</td>
<td>TAB CTRL+I</td>
</tr>
<tr>
<td>TOP–COLUMN</td>
<td></td>
<td>ESC CTRL+T</td>
<td></td>
</tr>
<tr>
<td>UNIX–END</td>
<td></td>
<td>CTRL+\</td>
<td></td>
</tr>
</tbody>
</table>

**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.
Handling User Input

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 chapter on colors and fonts in this book and the chapter on user interface environments in *OpenEdge Deployment: Managing 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.

```plaintext
i-action.p

ON F1 GO.
ON F2 HELP.
ON CTRL-X BELL.

FOR EACH Customer FIELDS(CustNum CreditLimit Name SalesRep) NO-LOCK:
   DISPLAY Customer.CustNum.
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 PROTERMCP 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 Figure 4–4.

<table>
<thead>
<tr>
<th>Syntax:</th>
<th>ON statement</th>
<th>Keyboard key</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ON F1 GO.</td>
<td>F1</td>
<td>GO</td>
</tr>
<tr>
<td></td>
<td>ON F2 HELP.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ON CTRL-X BELL.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Meaning: When the user presses ... this key ... takes this action.

**Figure 4–4:** 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. Table 4–5 lists these actions.

**Table 4–5: Actions you assign to keys**

<table>
<thead>
<tr>
<th>Action</th>
<th>Default key</th>
</tr>
</thead>
<tbody>
<tr>
<td>BACKSPACE</td>
<td>BACKSPACE</td>
</tr>
<tr>
<td>BACK–TAB</td>
<td>SHIFT+TAB, CTRL+U, CODE+TAB</td>
</tr>
<tr>
<td>BELL</td>
<td></td>
</tr>
<tr>
<td>CLEAR</td>
<td>F8, CTRL+Z, CODE+Z</td>
</tr>
<tr>
<td>CURSOR–UP</td>
<td>×, CTRL+K</td>
</tr>
<tr>
<td>CURSOR–DOWN</td>
<td>±, CTRL+J</td>
</tr>
<tr>
<td>CURSOR–LEFT</td>
<td>?, CTRL+H</td>
</tr>
<tr>
<td>CURSOR–RIGHT</td>
<td>?, CTRL+L</td>
</tr>
<tr>
<td>DELETE–CHARACTER</td>
<td>DEL</td>
</tr>
<tr>
<td>ENDKEY</td>
<td></td>
</tr>
<tr>
<td>END–ERROR</td>
<td>F4, CTRL+E, CODE+E, ESC (Windows)</td>
</tr>
<tr>
<td>ERROR</td>
<td></td>
</tr>
<tr>
<td>GO</td>
<td>F2, CTRL+X, CODE+X</td>
</tr>
<tr>
<td>HELP</td>
<td>F1, HELP, CTRL+W, CODE+W</td>
</tr>
<tr>
<td>HOME</td>
<td>HOME, ESC H (UNIX)</td>
</tr>
<tr>
<td>INSERT–MODE</td>
<td>F3, INSERT, CTRL+T, OVERTYPE</td>
</tr>
<tr>
<td>RECALL</td>
<td>F7, CTRL+R, CODE+R</td>
</tr>
<tr>
<td>RETURN</td>
<td>RETURN</td>
</tr>
<tr>
<td>STOP</td>
<td>CTRL+BREAK (Windows), CTRL+C (UNIX)</td>
</tr>
<tr>
<td>TAB</td>
<td>TAB, CTRL+I</td>
</tr>
</tbody>
</table>
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. Table 4–6 shows the mappings between the ABL portable mouse buttons and the physical mouse buttons in Windows.

<table>
<thead>
<tr>
<th>Portable button</th>
<th>Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT</td>
<td>LEFT mouse button</td>
</tr>
<tr>
<td>EXTEND</td>
<td>CTRL with LEFT mouse button</td>
</tr>
<tr>
<td>MENU</td>
<td>RIGHT mouse button</td>
</tr>
<tr>
<td>MOVE</td>
<td>LEFT mouse button</td>
</tr>
</tbody>
</table>

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 Table 4–6 (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 OpenEdge Development: 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.

![Diagram](image)

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.

```plaintext
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:
    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

```plaintext
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 Figure 4–5 appears.

![Figure 4–5: The i-gon1.p procedure](image)

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**

```fortran
FORM
   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:
   Customer.State
   WITH CENTERED FRAME upd-frame.
END.

ON F10 ANYWHERE DO:
   APPLY "GO" TO FRAME upd-frame.
   FIND NEXT Customer.
   Customer.City Customer.State
   WITH CENTERED FRAME upd-frame.
END.

ON F12 ANYWHERE DO:
   APPLY "GO" TO FRAME upd-frame.
   FIND PREV Customer.
   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

   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
- User interface triggers

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

```plaintext
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.
        Customer.State Customer.SalesRep EDITING:
            READKEY.
            IF FRAME-FIELD = "salesrep" AND LASTKEY = KEYCODE("F6") THEN DO:
                FOR EACH SalesRep FIELDS (SalesRep) NO-LOCK:
                    DISPLAY SalesRep.SalesRep WITH NO-LABELS 9 DOWN COLUMN 60 ROW 5.
            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.
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**

```plaintext
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.
        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.
                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.
END.

ON ANY-PRINTABLE OF Customer.SalesRep DO:
    BELL.
    RETURN NO-APPLY.
END.

REPEAT WITH FRAME cust-frame:
    PROMPT-FOR Customer.CustNum.
    FIND SalesRep OF Customer NO-LOCK.
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.
Monitoring keystrokes during data entry

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**

```plaintext
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.
    MESSAGE "Press F6 to see the previous Customer.").
        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.
                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

```pascal
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.
  Customer.State Customer.SalesRep
  WITH FRAME cust-frame.
END. /* ON F6 */

REPEAT WITH FRAME cust-frame:
  PROMPT-FOR Customer.CustNum.
  MESSAGE "Press F6 to see the previous Customer.".
  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 trigger 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:

```pascal
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.
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.

This chapter discusses input/output in 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 Figure 5–1.

Figure 5–1: 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

Figure 5–2 shows the output stream being redirected to a printer using the PRINTER option of the OUTPUT TO statement.

![Diagram showing output stream being redirected to a printer](image)

**Figure 5–2: 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 the “Stream I/O vs. screen I/O” section on page 5–9 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.
    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 the “Stream I/O vs. screen I/O” section on page 5–9 for more information.
Changing the output destination

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

```abl
/*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:
    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 Figure 5–2 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 the “Sending output to multiple destinations” section on page 5–7.
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 the “Sending output to multiple destinations” section on page 5–7. For more information on using the `CLIPBOARD` system handle for output (and also input), see Chapter 9, “System Clipboard.”

**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 the “Sending output to multiple destinations” section on page 5–7.
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.

```
DEFINE VARIABLE outfile AS CHARACTER FORMAT "x(8)"
   LABEL "Output file name ".

getfile:
   DO ON ERROR UNDO , RETRY:
      SET outfile WITH SIDE -LABELS.
      IF SEARCH outfile = outfile THEN DO:
         MESSAGE "A file named" outfile "already exists ".
         MESSAGE "Please use another name ".
         BELL.
         UNDO getfile , RETRY getfile.
      END.
   END.

OUTPUT TO VALUE (outfile).

FOR EACH customer:
   DISPLAY name credit -limit
      WITH NO-BOX NO-LABELS STREAM-IO.
END.
```

Figure 5–3: Multiple output destinations

In the procedure in Figure 5–3, 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 Figure 5–4 uses the `OUTPUT CLOSE` statement to reset the output destination from a file to the terminal.

```
i-out.p
OUTPUT TO cust.dat.
  FOR EACH customer:
    DISPLAY cust-num name address address2 city state
    SKIP(2) WITH 1 COLUMN SIDE-LABELS STREAM-IO.
  END.
OUTPUT CLOSE.
DISPLAY "Finished".
```

**Figure 5–4:** The `OUTPUT CLOSE` statement

The procedure in Figure 5–4 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.p1 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**

```plaintext
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 Table 5–1. 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 5–1: Printing options for Windows**

<table>
<thead>
<tr>
<th>Printing option</th>
<th>Value</th>
<th>Selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Context</td>
<td>0</td>
<td>Default print context</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td><strong>Print</strong> dialog box. This allows the user to establish a print context</td>
</tr>
<tr>
<td>Landscape orientation</td>
<td>2</td>
<td>Landscape orientation</td>
</tr>
</tbody>
</table>
Table 5–1: Printing options for Windows

<table>
<thead>
<tr>
<th>Printing option</th>
<th>Value</th>
<th>Selection</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paper Size</td>
<td>4</td>
<td>Letter</td>
</tr>
<tr>
<td></td>
<td>8</td>
<td>Legal</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td>A4</td>
</tr>
<tr>
<td>Paper Tray</td>
<td>32</td>
<td>Upper tray</td>
</tr>
<tr>
<td></td>
<td>64</td>
<td>Middle tray</td>
</tr>
<tr>
<td></td>
<td>96</td>
<td>Lower tray</td>
</tr>
<tr>
<td></td>
<td>128</td>
<td>Manual</td>
</tr>
<tr>
<td></td>
<td>160</td>
<td>Auto</td>
</tr>
<tr>
<td>UTF-8</td>
<td>512</td>
<td>Required when sending UTF-8 encoded data to the printer</td>
</tr>
</tbody>
</table>

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**

```plaintext
/*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 adecmm/_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 *OpenEdge Development: 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” section on page 5–29 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 Figure 5–5.

```
i-chgin.p

INPUT FROM i -datf1.d.
REPEAT:
CREATE customer .
SET cust-num name sales -rep.
END.
```

Figure 5–5: 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 *OpenEdge Deployment: Managing 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 the “Input from files” section on page 5–13.

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 the “Reading the contents of a directory” section on page 5–25.

You can reset any input source to the terminal using the TERMINAL option of the INPUT FROM statement. For more information, see the “Receiving input from multiple sources” section on page 5–14.

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.

\[ \text{FOR EACH Customer WHERE Customer.CustNum > 89:} \]
\[ \hspace{1em} \text{DELETE Customer.} \]
\[ \text{END.} \]
Figure 5–6 shows the i-chgin2.p procedure.

```plaintext
i-chgin2.p

DEFINE VARIABLE cust-num-var LIKE customer.cust-num.
DEFINE VARIABLE name-var LIKE customer.name.
DEFINE VARIABLE sales-rep-var LIKE customer.sales-rep.
DEFINE VARIABLE answer AS LOGICAL.

DISPLAY "The customers in the data file are: " WITH NO-BOX.

INPUT FROM p-datf1.d.

REPEAT WITH 10 DOWN COLUMN 38:
   SET cust-num-var name-var sales-rep-var WITH NO-BOX.
END.

INPUT FROM TERMINAL.

SET answer LABEL
   'Do you want to create database records for these customers?'
   WITH SIDE-LABELS NO-BOX FRAME ans-frame.
IF answer
   THEN DO:
      DISPLAY "Creating records for..." WITH FRAME ans-frame.
      INPUT FROM p-datf1.d.
      REPEAT:
         CREATE customer.
      END.
   END.
END.
```

Figure 5–6: 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. Figure 5–7 illustrates this scenario.

![Diagram](image)

**Figure 5–7: 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.

**Notes:** For information regarding the maximum number of streams allowed in OpenEdge applications, see the Input/Output Limits table in *OpenEdge Deployment: Managing ABL Applications*.

For information about the size limit of operating system files, see the input/output limits section of *OpenEdge Deployment: Managing ABL Applications*.
The procedure `i-dfstr.p` uses the two report scenarios shown in Figure 5-7.

**i-dfstr.p**

```plaintext
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 `exc` 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 `exc` 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 `exc` 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 `exc` in the previous `DISPLAY` statement. That means that the `exc` frame already contains the `exc` 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 \texttt{SHARED} parameter of the \texttt{DEFINE STREAM} statement to create a shared stream.

\textbf{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 the “Using stream object handles” section on page 5–21 for more information.

For example, the procedures \texttt{i-sstrm.p} and \texttt{i-dispho.p} share the same output stream, \texttt{phonelist}. Notice that \texttt{phonelist} is defined as a shared stream in both procedures.

\textbf{i-sstrm.p}

\begin{verbatim}
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.
\end{verbatim}

The \texttt{i-sstrm.p} procedure defines a \texttt{NEW SHARED STREAM} called \texttt{phonelist}. The procedure sends the output from the \texttt{phonelist} stream to a file called \texttt{phonefile}. The procedure also calls the \texttt{i-dispho.p} procedure.

\textbf{i-dispho.p}

\begin{verbatim}
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.
\end{verbatim}

The \texttt{i-dispho.p} procedure defines the \texttt{SHARED STREAM} \texttt{phonelist}, and displays the information from that stream on the screen. (It is more efficient to place the \texttt{FOR EACH} and \texttt{DISPLAY} statements in the \texttt{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 *OpenEdge Development: 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:

```abl
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:

```abl
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 *OpenEdge Development: ABL Reference*.
Summary of opening and closing streams

Table 5–2 describes how you establish, open, use, and close default streams and streams you name.

<table>
<thead>
<tr>
<th>Action</th>
<th>Unnamed streams</th>
<th>Named streams</th>
</tr>
</thead>
<tbody>
<tr>
<td>Establish the stream</td>
<td>By default, each procedure gets one unnamed input stream and one unnamed output stream.</td>
<td>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.</td>
</tr>
<tr>
<td>Open the stream</td>
<td>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 using OUTPUT TO or INPUT FROM.</td>
<td>You open the stream explicitly by using: • OUTPUT {STREAM \textit{stream}</td>
</tr>
<tr>
<td>Use the stream</td>
<td>All data-handling statements use the stream by default.</td>
<td>Use the name or the handle of the opened stream in the data-handling statement that will use the stream.</td>
</tr>
<tr>
<td>Close the stream</td>
<td>Automatically closed at the end of the procedure that opened it. You can also explicitly close it with the OUTPUT CLOSE or INPUT CLOSE statement.</td>
<td>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.</td>
</tr>
</tbody>
</table>

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 *OpenEdge Development: 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 *OpenEdge Deployment: Managing 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 OpenEdge Deployment: 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 OpenEdge Development: 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.

  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 OpenEdge Development: 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 OpenEdge Development: Internationalizing 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 OpenEdge Development: Internationalizing Applications.
For example, the `i-codpag.p` procedure writes to two output streams using two different code pages.

### i-codpag.p

```abl
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 *OpenEdge Development: 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 the “Input from files” section on page 5–13.

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 Appendix D, “Command and Utility Reference.”
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 (list of procedures a requesting procedure can access) of an OpenEdge AppServer. For more information on OpenEdge AppServers, see OpenEdge Application Server: Developing AppServer Applications.

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:
END.
OUTPUT CLOSE.
```

The output from i-export.p is written to i-datfl6.d.

```
i-datfl6.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-exp2t.p` writes to a file in which field values are separated by commas.

### i-exp2t.p

```abl
OUTPUT TO i-datf17.d.
FOR EACH Customer NO-LOCK:
END.
OUTPUT CLOSE.
```

The output from `i-exp2t.p` is written to `i-datf17.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 *OpenEdge Development: 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

```abl
OUTPUT TO i-datf18.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-datf18.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 *OpenEdge Development: 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

```abl
INPUT FROM i-datfl6.d.
REPEAT:
  CREATE Customer.
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 the “Using the EXPORT statement” section on page 5–30.

i-imprt2.p

```abl
INPUT FROM i-datfl7.d.
REPEAT:
  CREATE customer.
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**

```plaintext
INPUT FROM i-datf12.d.
REPEAT:
  CREATE Customer.
  IMPORT Customer.CustNum.
  IMPORT UNFORMATTED Customer.Name.
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 i-datf13.d.

**i-datf13.d**

```
90 Wind Chill Hockey  BBB
91 Low Key Checkers   DKP
92 Bing's Ping Pong   SLS
```

In i-datf13.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**

```plaintext
DEFINE VARIABLE file-line AS CHARACTER NO-UNDO.

INPUT FROM i-datf13.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.

```plaintext
i-datfl4.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.

```plaintext
i-impun3.p
```

```
INPUT FROM i-datfl4.d.
REPEAT:
  CREATE Customer.
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 *OpenEdge Development: ABL Reference.*
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. This chapter describes:

- 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 chapter that discusses Active X controls in Chapter 16, “ActiveX Control Support.”

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 OpenEdge Deployment: Managing ABL Applications.
OpenEdge default colors

The environment that comes with your OpenEdge installation specifies 16 colors. Table 6–1 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 6–1: OpenEdge default colors

<table>
<thead>
<tr>
<th>Color number</th>
<th>Windows RGB values</th>
<th>Actual color defined</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0, 0, 0</td>
<td>Black</td>
</tr>
<tr>
<td>1</td>
<td>0, 0, 128</td>
<td>Dark blue</td>
</tr>
<tr>
<td>2</td>
<td>0, 128, 0</td>
<td>Dark green</td>
</tr>
<tr>
<td>3</td>
<td>0, 128, 128</td>
<td>Blue green</td>
</tr>
<tr>
<td>4</td>
<td>128, 0, 0</td>
<td>Red</td>
</tr>
<tr>
<td>5</td>
<td>128, 0, 128</td>
<td>Purple</td>
</tr>
<tr>
<td>6</td>
<td>128, 128, 0</td>
<td>Olive</td>
</tr>
<tr>
<td>7</td>
<td>128, 128, 128</td>
<td>Gray</td>
</tr>
<tr>
<td>8</td>
<td>192, 192, 192</td>
<td>Light gray</td>
</tr>
<tr>
<td>9</td>
<td>0, 0, 255</td>
<td>Blue</td>
</tr>
<tr>
<td>10</td>
<td>0, 255, 0</td>
<td>Green</td>
</tr>
<tr>
<td>11</td>
<td>0, 255, 255</td>
<td>Turquoise</td>
</tr>
<tr>
<td>12</td>
<td>255, 0, 0</td>
<td>Red</td>
</tr>
<tr>
<td>13</td>
<td>255, 0, 255</td>
<td>Pink</td>
</tr>
<tr>
<td>14</td>
<td>255, 255, 0</td>
<td>Yellow</td>
</tr>
<tr>
<td>15</td>
<td>255, 255, 255</td>
<td>White</td>
</tr>
</tbody>
</table>
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 the “Color in character interfaces” section on page 6–9.

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**

```plaintext
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

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 OpenEdge Development: ABL Reference. For more information on using the COLOR–TABLE and FONT–TABLE handles for run-time color and font management, see the “Accessing the current color and font tables” section on page 6–19.
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 ActiveX Automation 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 Chapter 15, “ActiveX Automation Support,” and Chapter 16, “ActiveX Control Support.”
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.

For more information on these attributes and options, see *OpenEdge Development: ABL Reference.*
**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.

Table 6–2 shows how ABL assigns colors based on the state and category of widget.

**Table 6–2: Colors used in character interfaces**

<table>
<thead>
<tr>
<th>Widget state</th>
<th>Text input widget</th>
<th>Selectable widget</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insensitive</td>
<td><code>DCOLOR</code></td>
<td><code>DCOLOR</code></td>
</tr>
<tr>
<td>Sensitive</td>
<td><code>PFCOLOR</code></td>
<td><code>DCOLOR</code></td>
</tr>
<tr>
<td>Focus</td>
<td><code>PFCOLOR</code></td>
<td><code>PFCOLOR</code></td>
</tr>
</tbody>
</table>

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 PROTERMCP file, you can specify additional application color pairs from color 5 to 127. For more information, see *OpenEdge Deployment: Managing 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 the “Managing application environments” section on page 6–23.

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:

1. **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 the “Allowing the user to change colors and fonts” section on page 6–13.

2. **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 the “Accessing the current color and font tables” section on page 6–19.

3. **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 the “Retrieving and changing color and font definitions” section on page 6–21.

4. **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 the “Managing application environments” section on page 6–23.
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:

```plaintext
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 OpenEdge Development: ABL Reference and the “Accessing the current color and font tables” section on page 6–19.

Color dialog box

Figure 6–1 shows the Windows system color dialog box.

![Color dialog box](image)

**Figure 6–1:** 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**

```plaintext
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) SPACE(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:

![Choose frame colors](image)

If you click **OK**, a color dialog box appears (see Figure 6–1) 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

Figure 6–2 shows the native Windows system **Font** dialog box.

![Font dialog box](image)

**Figure 6–2:** 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 \texttt{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.

\texttt{i-fdial1.p}

\begin{verbatim}
IF FONT-TABLE:NUM-ENTRIES < 13 THEN 

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.
\end{verbatim}

When you run this procedure in Windows, the following dialog box appears:

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{fontdialog.png}
\caption{Choose frame fonts ...}
\end{figure}

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 \textbf{Radio Font} (font 11) or \textbf{Button Font} (font 12) radio item, then click \textbf{OK} to open the font dialog box shown in Figure 6–2. Click \textbf{CANCEL} to terminate the procedure without any further font changes.
To change the radio-set font selected in Figure 6–2 to bold 8-point Arial, set up the Font dialog as shown in Figure 6–3.

![Figure 6–3: Changing font 11 to bold 8-point Arial](image)

After you click OK, the Choose frame fonts dialog box reappears with the radio set in the new font. For example:

![Choose frame fonts](image)

**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:

```lisp
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:

```lisp
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:

```lisp
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 *OpenEdge Development: 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.
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."
```

For more information on the FONT–TABLE handle, see *OpenEdge Development: ABL Reference*.
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 the “Managing application environments” section on page 6–23).

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 the “Accessing the current color and font tables” section on page 6–19), 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 the “Allowing the user to change colors and fonts” section on page 6–13).

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 \texttt{PUT–KEY–VALUE} statement with the \texttt{COLOR} or \texttt{FONT} option. For example, if you allow the user to change color 8 during a session through the color common dialog (\texttt{SYSTEM–DIALOG COLOR} statement), you can save the new color definition in the current environment using this statement:

\begin{verbatim}
PUT–KEY–VALUE COLOR 8.
\end{verbatim}

You can save all current color definitions from the color table using this statement:

\begin{verbatim}
PUT–KEY–VALUE COLOR ALL.
\end{verbatim}

For more information on these statements, see \textit{OpenEdge Development: 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 *OpenEdge Development: 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.
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.

This chapter introduces you to the ABL (Advanced Business Language) statements that you can use to generate reports, both simple and complex, as described in the following sections:

- 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
Creating Reports

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:

```plaintext
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:

```plaintext
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:

```plaintext
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:

```plaintext
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:

```plaintext
FOR EACH Customer FIELDS(Balance Name) 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:

```
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 FIELDS(Balance Name) 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

The "Designing an interface for viewing report data" section on page 7–6 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 1-10-02.p and run it. The following screen appears:
2. Click Report. The Report Output dialog box appears:

![Report Output dialog box]

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
(1 of 2)

/********** 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 FIELDS (Balance Name) WITH STREAM-IO:
  lBalDue = IF Customer.Balance > 0 THEN TRUE ELSE FALSE.
  DISPLAY Customer.Name lBalDue.
END.
```
Creating Reports

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. This chapter covers directing output more fully 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.

/*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*/ ASSgn Rep-Editor:FONT = 3.
       ENABLE ALL WITH FRAME Frame1.
       WAIT-FOR CHOOSE OF b-exit.
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:

```plaintext
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.
Creating Reports

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 FIELDS(Name Comments) 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.
```

**Notes:** 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:

```
[ aggregate-option ] ... [ BY break-group ] ...
```

Table 7–1 lists the aggregate options.

### Table 7–1: Aggregate options

<table>
<thead>
<tr>
<th>Aggregate option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVERAGE</td>
<td>Calculates the average of all the expression’s values, and calculates the overall average for the break groups</td>
</tr>
<tr>
<td>SUB-AVERAGE</td>
<td>Calculates the average of all the expression’s values, but does not calculate the overall average</td>
</tr>
<tr>
<td>COUNT</td>
<td>Counts the number of items in the break group, and calculates the overall count of the break groups</td>
</tr>
<tr>
<td>SUB-COUNT</td>
<td>Counts the number of items in the break group, but does not calculate the overall count</td>
</tr>
<tr>
<td>MAXIMUM</td>
<td>Finds the maximum value of the expression in the break group, and finds the overall maximum of the break groups</td>
</tr>
<tr>
<td>SUB-MAXIMUM</td>
<td>Finds the maximum value of the expression in the break group, but does not find the overall maximum</td>
</tr>
<tr>
<td>MINIMUM</td>
<td>Finds the minimum value of the expression in the break group, and finds the overall minimum of the break groups</td>
</tr>
<tr>
<td>SUB-MINIMUM</td>
<td>Finds the minimum value of the expression in the break group, but does not find the overall minimum</td>
</tr>
</tbody>
</table>
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 FIELDS(Balance SalesRep) NO-LOCK
   BREAK 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:

![Report Output Table]

Table 7–1: Aggregate options

<table>
<thead>
<tr>
<th>Aggregate option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TOTAL</td>
<td>Calculates the total of all the expression’s values, and calculates the overall total for the break group.</td>
</tr>
<tr>
<td>SUB-TOTAL</td>
<td>Calculates the total of all the expression’s values, but does not calculate the overall total for the break groups</td>
</tr>
</tbody>
</table>

To see an example of control breaks and aggregate values:
Creating Reports

The members of each break group all have the same \texttt{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 \textbf{OK}, then \textbf{Exit}, and then press \texttt{SPACEBAR} to return to the Procedure Editor.

Here is the code for this procedure:

\texttt{\textit{i-10-05.p}}

```plaintext
/* \{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 FIELDS(Balance Name SalesRep) NO-LOCK
  BREAK BY Customer.SalesRep WITH STREAM-IO:
  DISPLAY Customer.SalesRep Customer.Name
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.
```

\textbf{Notes:} The \texttt{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.

```plaintext
i-10-06.p

... FOR EACH Customer FIELDS (Balance Contact CreditLimit Name) 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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Contact</th>
<th>Balance Cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Down Football</td>
<td>Marciia Shirley</td>
<td>76,904.00</td>
</tr>
<tr>
<td>UPV Yissimo</td>
<td>Hector D’Amato</td>
<td>7,052.00</td>
</tr>
<tr>
<td>Fallen Arch Running</td>
<td>Kevin Malone</td>
<td>60,716.00</td>
</tr>
<tr>
<td>Spike’s Volleyball</td>
<td>Craig Blesser</td>
<td>10,267.00</td>
</tr>
</tbody>
</table>

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

```plaintext
... FOR EACH Customer FIELDS(Balance Contact CreditLimit CustNum Name) 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 FIELDS(CustNum OrderNum OrderDate ShipDate PromiseDate)
       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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Contact</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Down</td>
<td>football</td>
<td>$76,934.00</td>
</tr>
<tr>
<td>Order-num: 3</td>
<td>Order</td>
<td></td>
</tr>
<tr>
<td>Shipped:</td>
<td>02/17/93</td>
<td></td>
</tr>
<tr>
<td></td>
<td>From:</td>
<td></td>
</tr>
<tr>
<td>Order-num: 65</td>
<td>Order</td>
<td></td>
</tr>
<tr>
<td>Shipped:</td>
<td>03/03/93</td>
<td></td>
</tr>
<tr>
<td></td>
<td>From:</td>
<td></td>
</tr>
<tr>
<td>Order-num: 80</td>
<td>Order</td>
<td></td>
</tr>
<tr>
<td>Shipped:</td>
<td>03/10/93</td>
<td></td>
</tr>
<tr>
<td></td>
<td>From:</td>
<td></td>
</tr>
<tr>
<td>Order-num: 199</td>
<td>Order</td>
<td></td>
</tr>
<tr>
<td>Shipped:</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>From:</td>
<td></td>
</tr>
</tbody>
</table>

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.
Creating Reports

The code fragment from `i-10-08.p` shows the completed structure of nested `FOR EACH` blocks.

### i-10-08.p

```plaintext
...  
FOR EACH Customer FIELDS(Balance CreditLimit Contact CustNum) 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 FIELDS(CustNum OrderDate OrderNum PromiseDate ShipDate) 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 FIELDS(ItemNum OrderNum Qty) 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:

<table>
<thead>
<tr>
<th>Qty</th>
<th>Item-num</th>
<th>Item-Name</th>
<th>Unit Price</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>95</td>
<td>00045</td>
<td>Golf shoes</td>
<td>81.00</td>
<td>$ 7,695.00</td>
</tr>
<tr>
<td>44</td>
<td>00054</td>
<td>Shin pads</td>
<td>4.86</td>
<td>$ 213.84</td>
</tr>
<tr>
<td>71</td>
<td>00004</td>
<td>Cycle Helmet</td>
<td>75.00</td>
<td>$ 5,325.00</td>
</tr>
<tr>
<td>99</td>
<td>00030</td>
<td>Windbreaker</td>
<td>32.75</td>
<td>$ 3,217.75</td>
</tr>
<tr>
<td>99</td>
<td>00002</td>
<td>Tennis Racque</td>
<td>64.50</td>
<td>$ 6,291.00</td>
</tr>
<tr>
<td>67</td>
<td>00032</td>
<td>Tennis Shorts</td>
<td>19.99</td>
<td>$ 1,339.33</td>
</tr>
</tbody>
</table>

$ 24,741.67  TOTAL

<table>
<thead>
<tr>
<th>Qty</th>
<th>Item-num</th>
<th>Item-Name</th>
<th>Unit Price</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>95</td>
<td>00024</td>
<td>Snorkel</td>
<td>12.95</td>
<td>$ 1,252.25</td>
</tr>
<tr>
<td>37</td>
<td>00007</td>
<td>Buoyancy Vest</td>
<td>125.00</td>
<td>$ 4,625.00</td>
</tr>
<tr>
<td>79</td>
<td>00023</td>
<td>Squash Balls</td>
<td>7.99</td>
<td>$ 631.21</td>
</tr>
<tr>
<td>44</td>
<td>00019</td>
<td>Wet Suit</td>
<td>225.00</td>
<td>$ 9,855.00</td>
</tr>
<tr>
<td>89</td>
<td>00018</td>
<td>Wet Suit</td>
<td>225.00</td>
<td>$ 20,022.50</td>
</tr>
<tr>
<td>11</td>
<td>00038</td>
<td>Shorts</td>
<td>13.99</td>
<td>$ 153.89</td>
</tr>
<tr>
<td>23</td>
<td>00040</td>
<td>Ice Skates</td>
<td>51.00</td>
<td>$ 1,173.00</td>
</tr>
</tbody>
</table>

Order-num: 3
Shipped: 02/17/93
Promised: 02/16/93

Order-num: 66
Shipped: 03/08/93
Promised: 03/09/93
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 OpenEdge Development: 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
```

Table 7–2 describes the OUTPUT statement syntax components.

<table>
<thead>
<tr>
<th>Table 7–2: OUTPUT statement options (1 of 2)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Component</td>
</tr>
<tr>
<td>STREAM</td>
</tr>
<tr>
<td>STREAM-HANDLE</td>
</tr>
<tr>
<td>PRINTER</td>
</tr>
<tr>
<td>opsys-file</td>
</tr>
</tbody>
</table>
Redirecting output

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.

```
/*1*/ OUTPUT TO "tut-temp.txt".
   FOR EACH Customer FIELDS(Balance Contact CreditLimit CustNum Name)
     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 FIELDS(CustNum OrderDate OrderNum PromiseDate ShipDate)
     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 FIELDS(ItemNum OrderNum Qty)
     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 OpenEdge Deployment: Startup Command and Parameter Reference.
Directing output to multiple destinations

ABL lets you specify multiple output destinations in a single procedure. You can use \texttt{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 \texttt{DEFINE STREAM} statement to create the streams you need. This is a partial syntax for \texttt{DEFINE STREAM}:

\begin{verbatim}
DEFINE STREAM stream-name
\end{verbatim}

To output to the stream, you reference it in your output statement with the keyword \texttt{STREAM}, as shown in the following example:

\begin{verbatim}
/*1*/ DEFINE STREAM sToFile.
/*2*/ OUTPUT TO PRINTER.
/*3*/ OUTPUT STREAM sToFile TO "tut-temp.txt".
    FOR EACH Customer FIELDS (Balance Name) 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.
\end{verbatim}

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 \texttt{OUTPUT TO} statement redirects the unnamed stream to the default printer.

- This \texttt{OUTPUT TO} statement redirects the named stream to output to a file.

- The first \texttt{DISPLAY} statement outputs to the unnamed stream only.

- The second \texttt{DISPLAY} statement outputs to the named stream only.

- This \texttt{OUTPUT CLOSE} statement redirects the default stream back to the screen.

- This \texttt{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:

```plaintext
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 OpenEdge Development: 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
```
Creating Reports

This code defines the first part of a procedure that implements the three-frame design:

```sql
/*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.

Customer Report continued on next page
```

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:

```
DEFINE FRAME f-ftr
    HEADER
    "Customer Report"
    "continued on next page"
/*1*/    WITH FRAME f-ftr PAGE-BOTTOM CENTERED STREAM-IO.
```

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

```plaintext
/*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 FIELDS(Balance SalesRep Name
   Contact CreditLimit Address City State PostalCode Phone)
   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)
   WITH PAGE-TOP FRAME f-hdr STREAM-IO.
/*5*/ DEFINE FRAME f-ftr
   HEADER
   "Customer Report"
   "continued next page"
   WITH FRAME f-ftr PAGE-BOTTOM CENTERED STREAM-IO.
/*6*/ VIEW FRAME f-hdr.
   VIEW FRAME f-ftr.
/*7*/ DISPLAY Name Balance Contact CreditLimit Address
   (City + ", " + State + ", " + PostalCode) @ Holder Phone
   WITH FRAME f-body.
/*8*/ IF LAST-OF(SalesRep) THEN DO:
   HIDE FRAME f-ftr.
   PAGE.
   END.
/*9*/ END. /* FOR EACH Customer */
/*10*/ OUTPUT CLOSE.
```
The following notes summarize the techniques shown in this chapter:

- 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.
Creating Reports

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:

```
Report Output

Michael Traicser
Hoops Croquet Co.
Suite 415
40 Grove St.
Kingman  MA  02111

Gloria Shepsey
Lift Line Skiing
276 North Street
Boston  MA  02114

Marcia Shanker
First Down Football
514 Edmonds Ave
Minneapolis  MN  55301

Robert Johns

OK
```

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, postalcode 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-in.i} /* Common Interface Setup Code */

/******* DEFINE TRIGGERS *******/
ON CHOOSE of b-rep DO:

/*1*/ OUTPUT TO "tut-temp.txt".
    FOR EACH Customer FIELDS (Address Address2 Balance City Contact Name
    PostalCode State) WHERE Customer.Balance >= 1400
    BY Customer.PostalCode:
    /*2*/ PUT
    Customer.Contact SKIP
    Customer.Name SKIP
    Customer.Address SKIP
    Customer.Address2 SKIP
    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`

```plaintext
{[i-10-in.i] /* Common Interface Setup Code */

/********** DEFINE TRIGGERS **********/
ON CHOOSE of b-rep DO:
  OUTPUT TO "tut-temp.txt".
  FOR EACH Customer FIELDS(Address Address2 Balance City Contact Name PostalCode State) 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.
```

7–33
The following is the output of `i-10-12.p`:

![Report Output]

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**

```plaintext
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**

```plaintext
/******* 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 FIELDS(Balance Name Phone SalesRep) NO-LOCK
        WHERE Customer.Balance >= 1400 WITH STREAM-IO:
            DISPLAY Customer.Name Customer.Phone Customer.Balance
    END.
    OUTPUT CLOSE.
END.

/*4*/ ON CHOOSE OF b-compress DO:
    OUTPUT TO PRINTER.
    PUT CONTROL Start-Compress.
    FOR EACH Customer FIELDS(Balance Name Phone SalesRep) NO-LOCK
        WHERE Customer.Balance >= 1400 WITH STREAM-IO:
            DISPLAY Customer.Name Customer.Phone Customer.Balance
    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 *OpenEdge Deployment: Managing ABL Applications* for more information about using the PUT CONTROL statement with control sequences.
Summary

This chapter 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-cause 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
Part III

External Program Interfaces

Chapter 8, Introduction to External Program Interfaces
Chapter 9, System Clipboard
Chapter 10, Providing Help for OpenEdge Applications
Chapter 11, Named Pipes
Chapter 12, Shared Library and DLL Support
Chapter 13, Windows Dynamic Data Exchange
Chapter 14, Using COM Objects in ABL
Chapter 15, ActiveX Automation Support
Chapter 16, ActiveX Control Support
Chapter 17, Sockets
Chapter 18, Host Language Call Interface
Introduction to External Program Interfaces

This chapter introduces 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.

This chapter contains the following sections:

- 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:

- The “UNIX shared library and Windows DLL support” section on page 8–14, and Chapter 12, “Shared Library and DLL Support”
- The “Sockets” section on page 8–25, and Chapter 17, “Sockets”
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.

Note: In versions of OpenEdge prior to Progress Version 9.1, assignment from one MEMPTR variable to another created a copy of the pointer, not the data. Thus, both variables referred to the same data after the assignment. Starting with Version 9.1, each variable refers to a completely separate copy of the data. However, a MEMPTR parameter passed to a local procedure continues to be passed by reference, and a MEMPTR parameter passed to a remote procedure is always passed by value (the same as an assignment statement).

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:** When working with shared library routines, you must fully understand the memory management provided by these routines before using MEMPTR variables with them effectively.
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 the “Initializing and uninitializing MEMPTR variables” section on page 8–3.

This is the syntax for the MEMPTR memory-writing statements (except PUT-STRING):

Syntax

\[
\text{PUT-datatype} \ ( \text{mptr-name} \ , \ \text{byte-position} ) = \text{expression}
\]

This is the syntax for the MEMPTR memory-reading functions (except GET-STRING and GET-BYTES):

Syntax

\[
\text{GET-datatype} \ ( \text{mptr-name} \ , \ \text{byte-position} )
\]

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-data type statements shown in Table 8–1.

Table 8–1: PUT-data type statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PUT-BYTE</td>
<td>Writes an INTEGER value to the specified 1-byte location</td>
</tr>
<tr>
<td>PUT-SHORT</td>
<td>Writes an INTEGER value to the specified 2-byte location</td>
</tr>
<tr>
<td>PUT-UNSIGNED-SHORT</td>
<td>Writes an unsigned INTEGER value to the specified 2-byte location</td>
</tr>
<tr>
<td>PUT-LONG</td>
<td>Writes an INTEGER value to the specified 4-byte location</td>
</tr>
<tr>
<td>PUT-FLOAT</td>
<td>Writes a decimal value to the specified 4-byte location as a single-precision floating-point value</td>
</tr>
<tr>
<td>PUT-DOUB E</td>
<td>Writes a decimal value to the specified 8-byte location as a double-precision floating-point value</td>
</tr>
<tr>
<td>PUT-STRING</td>
<td>Writes a character string value to the specified location, either null terminated or for a specified number of bytes</td>
</tr>
<tr>
<td>PUT-BYTES</td>
<td>Writes the contents of a RAW or MEMPTR variable to the specified byte location of a RAW or MEMPTR variable</td>
</tr>
</tbody>
</table>

For more information on these statements, see *OpenEdge Development: ABL Reference*. 
Memory-reading functions

ABL provides the GET-datatype functions shown in Table 8–2.

Table 8–2: GET-datatype functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET-BYTE</td>
<td>Returns the INTEGER value of the specified 1-byte location</td>
</tr>
<tr>
<td>GET-SHORT</td>
<td>Returns the INTEGER value of the specified 2-byte location</td>
</tr>
<tr>
<td>GET-UNSIGNED-SHORT</td>
<td>Returns the value of the specified 2-byte location, interpreted as an unsigned INTEGER</td>
</tr>
<tr>
<td>GET-UNSIGNED-LONG</td>
<td>Returns the unsigned 32-bit value at the specified memory location as an INT64</td>
</tr>
<tr>
<td>GET-INT64</td>
<td>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</td>
</tr>
<tr>
<td>GET-LONG</td>
<td>Returns the INTEGER value of the specified 4-byte location</td>
</tr>
<tr>
<td>GET-FLOAT</td>
<td>Returns the DECIMAL value of the specified 4-byte location, interpreted as a single-precision floating-point value</td>
</tr>
<tr>
<td>GET-DOUBLE</td>
<td>Returns the DECIMAL value of the specified 8-byte location, interpreted as a double-precision floating-point value</td>
</tr>
<tr>
<td>GET-STRING</td>
<td>Returns the CHARACTER string value from the specified location, either null terminated or for a specified number of bytes that can include nulls</td>
</tr>
<tr>
<td>GET-BYTES</td>
<td>Returns, as a MEMPTR or RAW value, the specified number of bytes from the specified byte location of a RAW or MEMPTR variable</td>
</tr>
</tbody>
</table>

For more information on these functions, see OpenEdge Development: ABL Reference.
**Copying between basic ABL data types and MEMPTR**

Table 8–3 lists the basic ABL data types and how you can copy them in and out of a MEMPTR variable.

<table>
<thead>
<tr>
<th>Data type</th>
<th>Copying to/from MEMPTR</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DATE</strong></td>
<td>To copy into a MEMPTR:</td>
</tr>
<tr>
<td></td>
<td>• Use INTEGER(date-expression) to obtain an integer value</td>
</tr>
<tr>
<td></td>
<td>• Use the PUT–LONG statement to copy integer value to MEMPTR</td>
</tr>
<tr>
<td></td>
<td>To copy from a MEMPTR:</td>
</tr>
<tr>
<td></td>
<td>• Use the GET–LONG function to return the integer value</td>
</tr>
<tr>
<td></td>
<td>• Use the DATE(integer-expression) function to return the date</td>
</tr>
<tr>
<td><strong>DECIMAL</strong></td>
<td>To copy into a MEMPTR, use the PUT–DOUBLE or PUT–FLOAT statement.</td>
</tr>
<tr>
<td><strong>INTEGER</strong></td>
<td>To copy into a MEMPTR, use the PUT–LONG, PUT–SHORT, PUT–UNSIGNED–SHORT, or PUT–BYTE statements.</td>
</tr>
<tr>
<td><strong>LOGICAL</strong></td>
<td>To copy into a MEMPTR, use the PUT–LONG, PUT–SHORT, PUT–UNSIGNED–SHORT, or PUT–BYTE statements.</td>
</tr>
<tr>
<td><strong>RAW</strong></td>
<td>To copy into a MEMPTR, assign the RAW value directly using an assignment statement or use the PUT–RAW statement.</td>
</tr>
<tr>
<td></td>
<td>To copy from a MEMPTR, assign the MEMPTR value directly using an assignment statement or use the GET–RAW function.</td>
</tr>
</tbody>
</table>

1. 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.
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**

\[
\text{PUT-BITS}( \text{destination}, \text{start-position}, \text{count} ) = \text{integer-expression}
\]

This statement interprets an integer (\text{integer-expression}) as the sequence of bits representing the binary value of \text{integer-expression}. For example, if the value of \text{integer-expression} is 22, the bit sequence is 10110. The statement interprets the INTEGER variable, \text{destination}, as a sequence of bits and writes the sequence of bits from \text{integer-expression} into \text{destination}, starting at the specified bit position (\text{start-position}). The bit position in \text{destination} is counted from the low-order bit, where the first bit is bit one (1). If the value of \text{integer-expression} is too large to store in the specified number of bits, ABL stores the low-order \text{count} bits of \text{integer-expression} in the specified \text{count} bits within \text{destination}.

The function to return some number of bits from an INTEGER variable, GET–BITS, has the following syntax:

**Syntax**

\[
\text{GET-BITS}( \text{source}, \text{start-position}, \text{count} )
\]

This function returns the INTEGER that represents the value of the number of bits specified by \text{count} starting at the specified low-order bit position (\text{start-position}) within the INTEGER variable specified by \text{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 *OpenEdge Development: 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 *OpenEdge Development: ABL Reference.*
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

Figure 8–1 shows the typical interactions between an ABL application and other graphical applications using the clipboard.

![Figure 8–1: ABL interactions using the system clipboard](image)

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

Figure 8–2 shows typical data exchanges between an OpenEdge application and an external application using named pipes.

Figure 8–2: 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. Chapter 11, “Named Pipes,” 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

Figure 8–3 shows a top-down structure diagram for an OpenEdge application that calls shared library functions.

---

**Figure 8–3:** 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 Chapter 12, “Shared Library and DLL Support.”
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 Chapter 14, “Using COM Objects in ABL.”

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

*Figure 8–4* shows a series of IPC exchanges between an OpenEdge application and another Windows application 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 spreadsheet or word-processing document. Depending on the server application, you might even be able to create new Automation object instances (new spreadsheets or documents) in the Automation Server from within your OpenEdge application.
Figure 8–5 shows a series of IPC exchanges between an OpenEdge application and the Automation objects of an Automation Server:

![ActiveX Automation Diagram]

**Figure 8–5: 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 8–4). 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.

Figure 8–6 shows how ABL supports ActiveX controls.

![Figure 8–6: ABL interface to ActiveX Controls](image)

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 Chapter 14, “Using COM Objects in ABL.”

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 Chapter 15, “ActiveX Automation Support.”

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 Chapter 16, “ActiveX Control Support.”
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 Chapter 14, “Using COM Objects in ABL.”

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 Chapter 16, “ActiveX Control Support.”

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 a `.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. For more information on accessing this list of events, see *OpenEdge Development: AppBuilder*.

**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 Chapter 14, “Using COM Objects in ABL.” For information specific to Automation objects, see Chapter 15, “ActiveX Automation Support,” and for information specific to ActiveX controls, see Chapter 16, “ActiveX Control Support.”

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 Secure Sockets Layer (SSL) connections. This support includes attributes for specifying the SSL configuration for an ABL client or ABL server SSL session and utilities for managing key and certificate stores for OpenEdge SSL clients and servers. For more information on SSL and OpenEdge SSL support, see OpenEdge Getting Started: Core Business Services.

Note: SSL 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, AppServer 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 the “Using MEMPTR to reference external data” section on page 8–2).
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 SSL clients

In addition to the basic ABL socket client capabilities, an ABL SSL client can:

- Access an OpenEdge-supported public key certificate store to properly validate communications with an SSL server (ABL or non-ABL)
- Turn off verification of the SSL 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 SSL servers

In addition to the basic ABL socket server capabilities, an ABL SSL 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 SSL clients (ABL or non-ABL)
- Determine how SSL sessions with clients are managed, including such automatic features as the time-out interval for idle 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.

- **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.)

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** — 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** — 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 I/O blocking or PROCESS EVENTS statement. Figure 8–7 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 8–7: ABL socket event model]

The server creates and enables a server socket object to allow Progress to listen for and accept connections 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 procedure in response to the CONNECT event. This procedure receives a handle to a socket object that is implicitly created on the server for the connection. Once a connection is established, both the client and server can read and write data to each other using their connected socket objects.

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 procedure that runs in response to a READ-RESPONSE event on the socket object. The client can specify this event procedure any time after it creates its socket object. The server can specify this event procedure any time after it receives the socket object in the CONNECT event procedure.
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.
- AppServer 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 SSL sessions, have access to a public key certificate store as maintained by the OpenEdge features for managing certificate stores. For more information, see *OpenEdge Getting Started: Core Business Services*.

For more information on programming with sockets, see Chapter 17, “Sockets.”
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 an 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 an OpenEdge application. For more information on how clipboard operations work in Windows, see the Microsoft Windows User’s Guide.

This chapter contains the following sections:

- 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 Table 9–1.

Table 9–1: CLIPBOARD handle attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Readable</th>
<th>Setable</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVAILABLE-FORMATS</td>
<td>CHARACTER</td>
<td>•</td>
<td>–</td>
</tr>
<tr>
<td>ITEMS-PER-ROW</td>
<td>INTEGER</td>
<td>•</td>
<td>●</td>
</tr>
<tr>
<td>MULTIPLE</td>
<td>LOGICAL</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>NUM-FORMATS</td>
<td>INTEGER</td>
<td>●</td>
<td>–</td>
</tr>
<tr>
<td>TYPE</td>
<td>CHARACTER</td>
<td>●</td>
<td>–</td>
</tr>
<tr>
<td>VALUE</td>
<td>CHARACTER</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>

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 OpenEdge Development: 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. (Note that in Windows, the clipboard can store a maximum of 64K of data). 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 *OpenEdge Development: 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 this chapter provides 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.
ELSE /* For FILL-IN */
    CLIPBOARD:VALUE = FOCUS:SCREEN-VALUE.
    FOCUS:SCREEN-VALUE = "".
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.
        END.
    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.".
        END.
    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
        ELSE
    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.)

```
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

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
            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 */
Single-item data transfers

i-clpbrd.p

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
    ELSE
  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 the “Single-item data transfers” section on page 9–4). 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-clpmul.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.

```pascal
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.Phone.
  CLIPBOARD:VALUE = STRING(Customer.CreditLimit).
END.

CLIPBOARD:MULTIPLE = FALSE.
```
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.

This chapter explains 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.

This chapter includes the following sections:

- 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.

Notes: 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 Figure 10–1.

![Figure 10–1: 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:

![Field Properties dialog box]

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 FIELDS(Name):
    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 Figure 10–2.

![Figure 10–2: Sample database field help text in the status area of a window](image)

**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:

```abl
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:

```abl
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 *OpenEdge Development: 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:

![Advanced Properties dialog box](image)

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:

![Example Window](image)

**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 Figure 10–3.

![AppBuilder](image)

**Figure 10–3: 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**

```plaintext
[ 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:

```plaintext
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:

```plaintext
SESSION:TOOLTIPS = FALSE
```

For more information on session attributes, see *OpenEdge Development: 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’s 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-TOC 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-TOPI 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-TOPI 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.

### Coding the help calling interface

This section explains some `SYSTEM-HELP` statement calls that are commonly used in the help calling interfaces of OpenEdge applications. For more information on the ABL elements described in the following sections, see *OpenEdge Development: ABL Reference*.

You can run the sample procedure, `r-syshlpchm.p`, to execute the help calls explained in this section. The source code example `r-syshlpchm.p` is in the Documentation and Samples directory (`doc_samples`) on the OpenEdge product DVD. You can also obtain source code examples from the OpenEdge Documentation page on PSDN:


This procedure demonstrates several features of the `SYSTEM-HELP` statement with the Procedure Editor help file (`editeng.chm`). The user can click a button to demonstrate each of the following `SYSTEM-HELP` options:

- **CONTEXT**
- **KEY**
- **ALTERNATE-KEY**
- **POSITION**
- **POSITION-MAXIMIZE**
- **QUIT**
To run r-syshlpchm.p:

1. Copy editeng.chm from \Program Files\Progress\OpenEdge\prohelp to your OpenEdge working directory (by default, C:\OpenEdge\WRK).

2. Open r-syshlpchm.p in the Procedure Editor.

3. Press F2 to run the file.

When you click the buttons on the sample interface, r-syshlpchm.p calls the Procedure Editor help file, editeng.chm.

```c
/* r-syshlpchm.p */

DEFINE VARIABLE helpfile AS CHARACTER NO-UNDO.
DEFINE BUTTON b_context LABEL "CONTEXT Call".
DEFINE BUTTON b_blank LABEL "KEY Call-''".
DEFINE BUTTON b_single LABEL "KEY Call-Tools".
DEFINE BUTTON b_full LABEL "KEY Call-Tools;Menu".
DEFINE BUTTON b_max LABEL "POSITION MAXIMIZE Call".
DEFINE BUTTON b_pos LABEL "POSITION Call".
DEFINE BUTTON b_alt LABEL "ALTERNATE-KEY Call".
DEFINE BUTTON b_quit LABEL "QUIT Call".

FORM
  SKIP(1) SPACE(1) b_context SPACE(1)
  SKIP(1) SPACE(1) b_blank SPACE(1)
  SKIP(1) SPACE(1) b_single SPACE(1)
  SKIP(1) SPACE(1) b_full SPACE(1)
  SKIP(1) SPACE(1) b_max SPACE(1)
  SKIP(1) SPACE(1) b_pos SPACE(1)
  SKIP(1) SPACE(1) b_alt SPACE(1)
  SKIP(1) SPACE(1) b_quit SPACE(1)
  SKIP(1) WITH FRAME x.
  ENABLE ALL WITH FRAME x.

helpfile = "editeng.chm".

/* The CONTEXT call displays the help topic associated with the specified context number of a help topic (in this case, 49256, for the Using Editor Buffers topic). */
ON CHOOSE OF b_context IN FRAME x DO:
  SYSTEM-HELP helpfile CONTEXT 49256.
END.

/* The KEY call brings up the topic matching the string found in the keyword index. If the string parameter is empty or is omitted altogether, the help viewer displays with the Index tab on top. */
ON CHOOSE OF b_blank IN FRAME x DO:
  SYSTEM-HELP helpfile KEY "".
END.
```
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 Figure 10–4.

![Figure 10–4: HTML Help Viewer showing Procedure Editor Help](image)
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.
```

Figure 10–5 shows the Help Viewer for the Procedure Editor help file with the Index tab on top.

![Help Viewer with Index tab on top](Figure 10–5: 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 Figure 10–6. When the user clicks the button, the Help Viewer displays the help topic for the current window or dialog box.

![Help button](Figure 10–6: 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 Figure 10–7.

![Figure 10–7: Context-sensitive help for the Buffer Information dialog box](image)

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:

```plaintext
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:

```plaintext
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 Figure 10–8. 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.”
Figure 10–8 depicts the behavior of the ABL HELP event.

**Figure 10–8: 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.
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.

This chapter contains the following sections:

- Overview of named pipes with ABL
- UNIX named pipes
- Windows named pipes
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).

Figure 11–1 shows a typical named pipe scenario.

![Diagram of a typical named pipe scenario](image)

**Figure 11–1: 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 11–1 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. Figure 11–2 illustrates this example.

Figure 11–2: 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 Figure 11–2, 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

This chapter provides 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 \texttt{mknod} command in the command line or the \texttt{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 the “UNIX named pipe examples” section on page 11–7.)

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

\textbf{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 \texttt{mknod} command on the command line or the \texttt{mknod()} system call from a C program. The two techniques produce the same results. The examples in this chapter 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

```bash
# 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:

```bash
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}
   ```
A C source file called `i-asksql.c` that implements the requestor:

```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);}
strncpy(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++ = '\\';
*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. Figure 11–3 illustrates this process.

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_RDONLY);
   ```
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-name SQL-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 \texttt{i-asksql.c}:

```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:

```c
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 \texttt{outpipe}. The requestor reads each line as it is written to the output pipe, as in the following statements from \texttt{i-asksql}. In the example, the requestor also writes each line to its standard output:

```c
char result[LEN];
int fdi, fdo, nread; int fdi, fdo, nread;
... 
while ((nread = read(fdo, result, LEN)) != 0)
{
    result[nread] = '\0';
    printf("%s", result);
}
```

\textbf{Note:} If there is no output, you might have entered your SQL statement incorrectly. This causes \texttt{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, \texttt{i-do-sq1.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:

```c
{1} WITH NO-LABELS.
```

```c
{1} WITH EXPORT.
```

10. The example requestor, \texttt{i-asksql}, continues to prompt for queries, repeating Actions 5 through 9, until you press \texttt{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 the “Operational characteristics of named pipes” section on page 11–3). 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. Table 11–1 lists these tasks and their C and ABL equivalents.

Table 11–1: Using C and ABL to access Windows named pipes

<table>
<thead>
<tr>
<th>Task</th>
<th>C</th>
<th>ABL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>CreateNamedPipe()</td>
<td>None</td>
</tr>
<tr>
<td>Connect</td>
<td>ConnectNamedPipe()</td>
<td>None</td>
</tr>
<tr>
<td>Read</td>
<td>ReadFile()</td>
<td>INPUT FROM</td>
</tr>
<tr>
<td></td>
<td>FlushFileBuffers()</td>
<td></td>
</tr>
<tr>
<td>Write</td>
<td>WriteFile()</td>
<td>OUTPUT TO</td>
</tr>
<tr>
<td>Close</td>
<td>CloseHandle()</td>
<td>INPUT CLOSE OUTPUT CLOSE</td>
</tr>
</tbody>
</table>

As Table 11–1 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:

<table>
<thead>
<tr>
<th>The C program . . .</th>
<th>While the ABL program . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Solicits a customer number from the user</td>
<td>• Reads the named pipe, getting the customer number the user entered</td>
</tr>
<tr>
<td>• Writes the customer number to the named pipe</td>
<td>• Retrieves the row of the customer with the specified customer number</td>
</tr>
<tr>
<td></td>
<td>• Displays the row of the customer table</td>
</tr>
</tbody>
</table>

If you tell the ABL program to write and the C program to read:

<table>
<thead>
<tr>
<th>The ABL Program . . .</th>
<th>While the C program . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Reads the entire customer table</td>
<td>• Reads the named pipe into a buffer</td>
</tr>
<tr>
<td>• Writes the name of each customer to the named pipe</td>
<td>• Displays the number of bytes read and the contents of the buffer</td>
</tr>
</tbody>
</table>

## 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-ablpip.p

```abl
/* 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 FIELDS(Name) 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.

```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("Waiting for connection...\n");
    if (ConnectNamedPipe(hPipe, NULL) == FALSE)
    { printf("Error connecting to named pipe, %ld", GetLastError());
        exit(GetLastError());
    }
    printf("and connection established\n");
```
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.
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. The following sections describe how to use shared libraries in your OpenEdge applications:

• 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 \texttt{PROCEDURE} statement.
  - For dynamic access use the methods and attributes of the call object handle. Create a call object with the \texttt{CREATE CALL} statement, then declare each routine using the call object’s \texttt{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 \texttt{DEFINE PARAMETER} statement
  - For dynamic access, parameters are defined with the call object’s \texttt{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 \texttt{RUN} statement, passing the specified fields, variables, and any required Windows widget handles
  - For dynamic access, use the call object’s \texttt{INVOKE( )} method

- **Build and manage any structures used by the routine**
Table 12–1 compares the ABL elements for declaring, accessing, and executing a shared library routine, both statically and dynamically. Options are shown in brackets.

### Table 12–1: Comparison of static and dynamic shared library access

<table>
<thead>
<tr>
<th>Task</th>
<th>Static</th>
<th>Dynamic (Call object handle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name the access point or shared library routine</td>
<td>PROCEDURE proc-name:</td>
<td>CALL-NAME = &quot;proc-name&quot;</td>
</tr>
<tr>
<td>Specify the library name</td>
<td>EXTERNAL &quot;dllname&quot;</td>
<td>LIBRARY = &quot;dllname&quot;</td>
</tr>
<tr>
<td>Specify the calling convention</td>
<td>[ CDECL</td>
<td>PASCAL</td>
</tr>
<tr>
<td>Specify the number of the entry point for the routine</td>
<td>[ ORDINAL n ]</td>
<td>[ ORDINAL = n ]</td>
</tr>
<tr>
<td>Indicate if the shared library remains in memory</td>
<td>[ PERSISTENT ]</td>
<td>[ PERSISTENT = {TRUE</td>
</tr>
<tr>
<td>Define the parameters</td>
<td>[DEFINE iomode PARAMETER parameter param-type.]</td>
<td>[SET-PARAMETER (parameter-number, data-type, iomode, parameter-value)]</td>
</tr>
<tr>
<td>Parameter iomode options</td>
<td>[INPUT</td>
<td>OUTPUT</td>
</tr>
<tr>
<td>Parameter param-type options</td>
<td>[ [LIKE field]</td>
<td>[AS [HANDLE TO ]data-type ] ]</td>
</tr>
<tr>
<td>Expected return data-type by shared library routine</td>
<td>[DEFINE RETURN PARAMETER parameter AS data-type]</td>
<td>[RETURN-VALUE-DLL-TYPE = &quot;data-type&quot;]</td>
</tr>
<tr>
<td>Execute the library routine</td>
<td>RUN proc-name[ (parameter-list)].</td>
<td>INVOKE( )</td>
</tr>
<tr>
<td>Unload the shared library</td>
<td>RELEASE EXTERNAL [PROCEDURE ]&quot;dllname&quot;.</td>
<td>RELEASE EXTERNAL &quot;dllname&quot;.</td>
</tr>
</tbody>
</table>

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
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 the “Using shared libraries” section on page 12–2.

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.

**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. Table 12–2 shows how to specify calling conventions.

Table 12–2: DLL calling conventions

<table>
<thead>
<tr>
<th>To specify this calling convention . . .</th>
<th>Code this option . . .</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C</td>
</tr>
<tr>
<td>Pascal</td>
<td>PASCAL</td>
</tr>
<tr>
<td>Standard</td>
<td>STDCALL</td>
</tr>
</tbody>
</table>

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.

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.
```
Accessing a shared library entry point

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**

\[
\text{SET-PARAMETER}(\text{parameter-number}, \text{data-type}, \text{iomode}, \text{parameter-value})
\]

*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 *OpenEdge Development: 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 12–4.

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:

\[
\begin{align*}
\text{hCall:NUM-PARAMETERS} & = 2. \\
\text{hCall:SET-PARAMETER}( 1, "LONG", "INPUT", msecs). \\
\text{hCall:SET-PARAMETER}( 2, "INTEGER", "INPUT", 5000).
\end{align*}
\]
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
- Passing arrays as parameters
- Passing the Unknown value and NULL values
- Using structure parameters
- Using MEMPTR variables as parameters
- DLL routines and ABL widgets (Windows only)

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 the “Using structure parameters” section on page 12–16.

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.
Table 12–3 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 12–3: C and DLL parameter data type compatibilities

<table>
<thead>
<tr>
<th>Example C data type</th>
<th>DLL parameter data type</th>
<th>Data type size and usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>BYTE</td>
<td>8-bit unsigned integer</td>
</tr>
<tr>
<td>short</td>
<td>SHORT</td>
<td>16-bit signed integer</td>
</tr>
<tr>
<td>unsigned short</td>
<td>UNSIGNED-SHORT</td>
<td>16-bit unsigned integer</td>
</tr>
<tr>
<td>int</td>
<td>LONG</td>
<td>32-bit signed integer</td>
</tr>
<tr>
<td>long (32-bit UNIX, Win32)</td>
<td>LONG</td>
<td>32-bit signed integer</td>
</tr>
<tr>
<td>unsigned int</td>
<td>UNSIGNED-LONG</td>
<td>32-bit unsigned integer</td>
</tr>
<tr>
<td>__int64 (Win32)</td>
<td>INT64</td>
<td>64-bit signed integer</td>
</tr>
<tr>
<td>long long (UNIX 32-bit)</td>
<td>INT64</td>
<td>64-bit signed integer</td>
</tr>
<tr>
<td>long (UNIX 64-bit)</td>
<td>INT64</td>
<td>64-bit signed integer</td>
</tr>
<tr>
<td>float</td>
<td>FLOAT</td>
<td>4-byte floating point</td>
</tr>
<tr>
<td>double</td>
<td>DOUBLE</td>
<td>8-byte floating point</td>
</tr>
<tr>
<td>char*</td>
<td>CHARACTER</td>
<td>Address (32 bits for 32-bit platforms, 64 bits for 64-bit platforms.)</td>
</tr>
<tr>
<td>c-data-type&lt;sup&gt;2&lt;/sup&gt;</td>
<td>HANDLE TO parameter-data-type&lt;sup&gt;2&lt;/sup&gt;</td>
<td>Address (32 bits for 32-bit platforms, 64 bits for 64-bit platforms.)</td>
</tr>
<tr>
<td>char*, output-pointer (which can be char**, short**, and so on), or a pointer to a structure.</td>
<td>MEMPTR</td>
<td>Address (32 bits for 32-bit platforms, 64 bits for 64-bit platforms.)</td>
</tr>
</tbody>
</table>

1. The C data type `int` generally specifies a size that depends on the operating system.
2. 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*`).

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.

**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:

```c
updateCounts(short * parray);
```

This example is the equivalent ABL for the updateCounts procedure when accessing the function statically:

```abl
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:

```abl
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 the “MEMPTR arrays” section on page 12–14.

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 "OpenEdge Development: 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 the “Using MEMPTR variables as parameters” section on page 12–16.

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 than a scalar value. ABL provides the DEFINE VARIABLE...AS MEMPTR statement. Once initialized, 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 Chapter 8, “Introduction to External Program Interfaces.” 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 parameters to a shared library routine

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 OpenEdge Development: 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
- Executing a shared library routine for dynamic access
- Parameter data types
- INT64 parameter conversions

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 the “Passing CHARACTER values to shared library routines” section on page 12–17 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. Table 12–4 lists each shared library parameter data type and its compatible ABL data type.

Table 12–4: Compatibility of shared library and ABL parameters

<table>
<thead>
<tr>
<th>Shared library parameter data types</th>
<th>ABL parameter data types</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE</td>
<td>INTEGER, INT64</td>
</tr>
<tr>
<td>SHORT</td>
<td>INTEGER, INT64</td>
</tr>
<tr>
<td>UNSIGNED-SHORT</td>
<td>INTEGER, INT64</td>
</tr>
<tr>
<td>LONG</td>
<td>INTEGER, INT64</td>
</tr>
<tr>
<td>UNSIGNED-LONG</td>
<td>INTEGER, INT64</td>
</tr>
<tr>
<td>INT64</td>
<td>INTEGER, INT64</td>
</tr>
<tr>
<td>FLOAT</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>DOUBLE</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>MEMPTR</td>
<td>MEMPTR</td>
</tr>
</tbody>
</table>

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 the “Passing CHARACTER values to shared library routines” section on page 12–17. For more information on passing a MEMPTR parameter, see the “Using MEMPTR variables as parameters” section on page 12–16.

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. Table 12–5 compares these options for statically and dynamically accessed shared libraries.

Table 12–5: Loading and unloading shared libraries

<table>
<thead>
<tr>
<th>You want to ...</th>
<th>Static</th>
<th>Dynamic (Call object handle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load and unload a shared library each time you invoke it</td>
<td>PROCEDURE statement without the PERSISTENT option</td>
<td>call-object:PERSISTENT = FALSE or do not specify the attribute</td>
</tr>
<tr>
<td>Load a shared library and keep it loaded</td>
<td>PROCEDURE statement with the PERSISTENT option</td>
<td>call-object:PERSISTENT = TRUE</td>
</tr>
<tr>
<td>Manually unload a shared library</td>
<td>RELEASE EXTERNAL statement</td>
<td>RELEASE EXTERNAL statement</td>
</tr>
</tbody>
</table>
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**
- **Examples of dynamic access to a shared library**

### 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-dlllex1.p
/* i-dlllex1.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-dlllex2.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-FIENALME.
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.
```

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:

```
/* 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.
```
PROCEDURE PaintRgn EXTERNAL "gdi32.dll":
  DEFINE INPUT PARAMETER DeviceHandle AS LONG.
  DEFINE INPUT PARAMETER RegionHandle AS LONG.
END PROCEDURE.

PROCEDURE GetDC EXTERNAL "user32.exe":
  DEFINE RETURN PARAMETER DeviceHandle AS LONG.
  DEFINE INPUT PARAMETER WindowHandle AS LONG.
END PROCEDURE.

PROCEDURE ReleaseDC EXTERNAL "user32.exe":
  DEFINE INPUT PARAMETER DeviceHandle AS LONG.
END PROCEDURE.

PROCEDURE DeleteObject EXTERNAL "gdi32.dll":
  DEFINE INPUT PARAMETER RegionHandle AS LONG.
END PROCEDURE.

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.

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.

DISPLAY "Preparing drawing...".
PAUSE.

RUN GetDC (OUTPUT hDevice, INPUT CURRENT-WINDOW:HWND).
RUN CreateEllipticRgnIndirect (OUTPUT hRegion, INPUT ElipRegion).
RUN GetStockObject (OUTPUT hObject, INPUT 4).
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:

```plaintext
/* 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.
```

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.

```abl
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.
```
Windows Dynamic Data Exchange

*Dynamic data exchange* (DDE) is a general protocol for inter-process communication in Windows.

This chapter contains the following sections:

- 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 Chapter 14, “Using COM Objects in ABL.”

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. Table 13–1 lists these DDE statements.

Table 13–1: DDE statements

<table>
<thead>
<tr>
<th>Statement</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDE ADVISE</td>
<td>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.</td>
</tr>
<tr>
<td>DDE EXECUTE</td>
<td>Sends one or more application commands to the DDE server to execute.</td>
</tr>
<tr>
<td>DDE GET</td>
<td>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.</td>
</tr>
<tr>
<td>DDE INITIATE</td>
<td>Opens a DDE conversation.</td>
</tr>
<tr>
<td>DDE REQUEST</td>
<td>Requests the current value of a server data item.</td>
</tr>
<tr>
<td>DDE SEND</td>
<td>Sends a new value to a server data item.</td>
</tr>
<tr>
<td>DDE TERMINATE</td>
<td>Closes a DDE conversation.</td>
</tr>
</tbody>
</table>

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 OpenEdge Development: 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.
Figure 13–1 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.

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 this chapter 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. Table 13–2 lists these DDE frame attributes.

Table 13–2: DDE frame attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Type</th>
<th>Readable</th>
<th>Setable</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDE-ERROR</td>
<td>INTEGER</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>DDE-ID</td>
<td>INTEGER</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>DDE-ITEM</td>
<td>CHARACTER</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>DDE-NAME</td>
<td>CHARACTER</td>
<td>✓</td>
<td>–</td>
</tr>
<tr>
<td>DDE-TOPIC</td>
<td>CHARACTER</td>
<td>✓</td>
<td>–</td>
</tr>
</tbody>
</table>

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 the “Exchanging data in conversations” section on page 13–12.)
DDE-ERROR

This attribute records the error condition returned by the last exchange for the frame. Table 13–3 lists the possible errors returned by DDE conversation exchanges.

Table 13–3: ABL DDE errors

<table>
<thead>
<tr>
<th>Error</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DDE INITIATE failure</td>
</tr>
<tr>
<td>2</td>
<td>A DDE statement (DDE ADVISE, EXECUTE, GET, REQUEST, or SEND) time-out</td>
</tr>
<tr>
<td>3</td>
<td>Memory allocation error</td>
</tr>
<tr>
<td>4</td>
<td>Invalid channel number (not an open conversation)</td>
</tr>
<tr>
<td>5</td>
<td>Invalid data item (in topic)</td>
</tr>
<tr>
<td>6</td>
<td>DDE ADVISE failure (data link not accepted)</td>
</tr>
<tr>
<td>7</td>
<td>DDE EXECUTE failure (commands not accepted)</td>
</tr>
<tr>
<td>8</td>
<td>DDE GET failure (data not available)</td>
</tr>
<tr>
<td>9</td>
<td>DDE SEND failure (data not accepted)</td>
</tr>
<tr>
<td>10</td>
<td>DDE REQUEST failure (data not available)</td>
</tr>
<tr>
<td>11</td>
<td>Data for DDE-NOTIFY event not available</td>
</tr>
<tr>
<td>99</td>
<td>Internal error (unknown)</td>
</tr>
</tbody>
</table>

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

```plaintext
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. */
```
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:

```i-ddeex2.p
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".
...```

/*
/*  0  System was out of memory, executable file was corrupt, or
/*       relocations were invalid. */
/*  1  Attempts were made to dynamically link to a task, or there
/*       was a sharing or network protection error. */
/*  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 requires Microsoft Windows 32-Bit extensions. */
**********************************************************************/
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 the “Initiating a conversation” section on page 13–11. 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 the “Closing DDE conversations” section on page 13–16.

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:

```plaintext
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 FIELDS(Balance Name State)
    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.

1. **To set up and manage event-driven exchanges for a data item:**
   - 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.
   - 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 OpenEdge Development: ABL Reference.

Applications for event-driven exchanges

You might use event-driven exchanges to tie data item values in an OpenEdge application to those in a spreadsheet, calendar 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 the “Defining conversation endpoints—DDE frames” section on page 13–7.

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):

```abl
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:

```plaintext
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)]".
  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.
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 "[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 FIELDS(Balance Name State) NO-UNDO
   WHERE Customer.Balance < 5000 BY Customer.Balance:
      itemn = "R" + STRING(rowi) + "C1".
      itemn = "R" + STRING(rowi) + "C2".
      DDE SEND sheet SOURCE STRING(Customer.Balance) ITEM itemn.
      RUN LogExchange.
      itemn = "R" + STRING(rowi) + "C3".
      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 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.
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).

This chapter describes the common features of COM objects supported by OpenEdge® and how to work with them in an ABL application and your development environment, as detailed in the following sections:

- 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

For an overview of the architectural elements that support COM objects in OpenEdge, see Chapter 8, “Introduction to External Program Interfaces.” For information on COM object support that is unique for Automation objects, see Chapter 15, “ActiveX Automation Support” and for ActiveX controls, see Chapter 16, “ActiveX Control Support.” The information in this chapter applies equally to Automation objects and ActiveX controls, except where noted.
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.
Table 14–1 compares each type of object (COM object and ABL widget) and summarizes how it is supported in ABL.

<table>
<thead>
<tr>
<th>Feature</th>
<th>COM object support</th>
<th>ABL widget support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compilation</td>
<td>No compile-time checking</td>
<td>Compile-time checking for attribute and method references</td>
</tr>
<tr>
<td>Instantiation</td>
<td>Dynamic, with different mechanisms for Automation objects and ActiveX controls</td>
<td>Static or Dynamic, depending on the widget and application</td>
</tr>
<tr>
<td>Handle type</td>
<td>COM-HANDLE</td>
<td>HANDLE</td>
</tr>
<tr>
<td>Handle validity</td>
<td>VALID-HANDLE function</td>
<td>VALID-HANDLE function</td>
</tr>
<tr>
<td>Access to data</td>
<td>Properties (optionally indexed)</td>
<td>Attributes</td>
</tr>
<tr>
<td>Access to behavior</td>
<td>Methods (with unnamed parameters)</td>
<td>Methods (with unnamed parameters)</td>
</tr>
<tr>
<td>Access to events</td>
<td>Event procedures (with unnamed parameters)</td>
<td>Triggers</td>
</tr>
<tr>
<td>Integration with ABL data</td>
<td>Extensive automatic mapping between COM data types and ABL data types</td>
<td>Full integration with ABL data types and ABL data management</td>
</tr>
<tr>
<td>Color management</td>
<td>RGB color values (RGB-VALUE function, GET-RGB-VALUE method on the COLOR-TABLE system handle)</td>
<td>Index numbers of RGB color values in ABL color table (COLOR-TABLE system handle)</td>
</tr>
<tr>
<td>Dynamic object management</td>
<td>RELEASE OBJECT statement</td>
<td>DELETE Widget statement</td>
</tr>
</tbody>
</table>

The remainder of this chapter describes 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:

```abl
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 Chapter 15, “ActiveX Automation Support.”

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 Chapter 16, “ActiveX Control Support.”

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 Chapter 15, “ActiveX Automation Support.” For information on ActiveX control event management, see Chapter 16, “ActiveX Control Support.”
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 OpenEdge Development: ABL Reference, but describe method and property references in a more top-down fashion. (See the information on attributes and methods in OpenEdge Development: 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 in the expression.

---

**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 14–2. 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 the “Specifying options for properties and method parameters” section on page 14–10.
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:

```abl
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:

```abl
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**

```abl
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 14–2. For more information on specifying the `AS datatype option` and data type conversion for properties, see the “Specifying options for properties and method parameters” section on page 14–10.

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:

```abl
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:

```abl
DEFINE VARIABLE myObject AS COM-HANDLE NO-UNDO.
/* ... Access initial COM object as myObject ... */
myObject:SavingsAccount("1000-24-369"):AccountName(1) = "John".
myObject:SavingsAccount("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 Table 14–2. For more information on mapping data types shown in Table 14–2, see Appendix A, “COM Object Data Type Mapping.”

**Table 14–2: Data-types for COM object methods and properties** (1 of 2)

<table>
<thead>
<tr>
<th>Data-type specifier to override the default mapping</th>
<th>ABL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CURRENCY</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>ERROR-CODE</td>
<td>INTEGER</td>
</tr>
<tr>
<td>IUNKNOWN</td>
<td>COM-HANDLE</td>
</tr>
<tr>
<td>FLOAT</td>
<td>DECIMAL</td>
</tr>
</tbody>
</table>
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 \texttt{datatype} or any of the other type options (\texttt{BY-POINTER} or \texttt{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 \texttt{OUTPUT} or \texttt{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 the “Specifying options for properties and method parameters” section on page 14–10, and Appendix A, “COM Object Data Type Mapping.”

### 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 the “Property and method syntax” section on page 14–5). 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 14–2 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 Appendix A, “COM Object Data Type Mapping.”)

- The type options \texttt{BY-POINTER} and \texttt{BY-VARIANT-POINTER} specify additional information for passing method parameters.

- The mode options \texttt{OUTPUT} and \texttt{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.

<table>
<thead>
<tr>
<th>Data-type specifier to override the default mapping</th>
<th>ABL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHORT</td>
<td>INTEGER</td>
</tr>
<tr>
<td>UNSIGNED-BYTE</td>
<td>INTEGER</td>
</tr>
</tbody>
</table>

Table 14–2: Data-types for COM object methods and properties (2 of 2)
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 Appendix A, “COM Object Data Type Mapping.”

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 the “Locating COM object information on your system” section on page 14–18.

**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 Appendix A, “COM Object Data Type Mapping.”

**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 14–2. For example:

```plaintext
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 14–2. 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:

```plaintext
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:

```plaintext
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 the “Locating COM object information on your system” section on page 14–18).

**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):

```abl
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 the “Managing COM object resources” section on page 14–16.

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 */

. . .
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 Chapter 8, “Introduction to External Program Interfaces.”). 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:

```abl
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.
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:

2. Click the COM Object Viewer icon in the PRO*Tools toolbar:

   ![COM Object Viewer](image)

   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 Figure 14–1.

![Automation objects in the COM Object Viewer](image)

Figure 14–1: 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 Chapter 15, “ActiveX Automation Support.”). 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 Figure 14–2.

Figure 14–2: 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 Chapter 16, “ActiveX Control Support.”) 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 14–2 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
ActiveX Automation allows one application (the Automation Controller) to drive another application (the Automation Server) 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.

This chapter describes 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 Chapter 14, “Using COM Objects in ABL.” This chapter contains the following sections:

- Requirements for doing Automation
- Accessing Automation Servers
- Managing Automation objects
- Automation event support
- Example Automation applications
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 OpenEdge Development: 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.
```

Figure 15–1 summarizes the basic logic for this option.

**Figure 15–1:** 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:

```vbnet
DEFINE VARIABLE hExcelObject AS COM-HANDLE NO-UNDO.
CREATE "Excel.Application" hExcelObject CONNECT.
```

Figure 15–2 summarizes the basic logic for this option.

![Diagram](attachment:image.png)

**Figure 15–2: Automation connection option 2**

**Note:** This option is equivalent to the Visual Basic GetObject(<i>class</i>) 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".
```

Figure 15–3 summarizes the basic logic for this option (ignoring the listed error conditions).

![Flowchart](#)

**Figure 15–3: 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".
```

Figure 15–4 summarizes the basic logic for this option (ignoring the listed error conditions).

![Diagram of 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 Chapter 14, “Using COM Objects in ABL.”
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**

```plaintext
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 Chapter 14, “Using COM Objects in ABL.”

For information on ActiveX control event management, see Chapter 16, “ActiveX Control Support.”
Example Automation applications

ABL comes installed with a number of sample Automation applications that you can use to test and borrow code for your own application development. These reside in separate subdirectories under %DLC%\src\samples\ActiveX. Each subdirectory contains a set of files for one application. These files include a readme.txt file that describes the requirements for running the application and the capabilities that it demonstrates.

For example, the ExcelGraphs subdirectory provides oleauto.p. This application creates an Excel bar chart that graphs sales data from the sports2000 database.

The following procedure listing shows oleauto.p. The bolded code shows the five component handles and where they are used to instantiate and release Automation objects. Only the Excel Application object is instantiated with the CREATE Automation Object statement. The rest are instantiated from methods of the Application object and its subordinate objects.
As the comments indicate, this procedure starts Excel, generates the graph from the sports2000 database, and exits leaving Excel and the graph open on your Windows desktop. Thus, this application really functions as a startup file for Excel and releases all of its instantiated Automation objects just prior to terminating. The objects that Excel requires remain instantiated, as shown:

```
/* This sample extracts data from an OpenEdge database and graphs the information using the Automation Objects from the Excel server in Microsoft Office. You must connect to a sports2000 database before running this. This sample program leaves Excel open. You should close it manually when the program completes. */
DEFINE VARIABLE chExcelApplication AS COM-HANDLE NO-UNDO.
DEFINE VARIABLE chWorkbook AS COM-HANDLE NO-UNDO.
DEFINE VARIABLE chWorksheet AS COM-HANDLE NO-UNDO.
DEFINE VARIABLE chChart AS COM-HANDLE NO-UNDO.
DEFINE VARIABLE chWorksheetRange AS COM-HANDLE NO-UNDO.
DEFINE VARIABLE iCount AS INTEGER NO-UNDO.
DEFINE VARIABLE iIndex AS INTEGER NO-UNDO.
DEFINE VARIABLE iTotalNumberOfOrders AS INTEGER NO-UNDO.
DEFINE VARIABLE iMonth AS INTEGER NO-UNDO.
DEFINE VARIABLE dAnnualQuota AS DECIMAL NO-UNDO.
DEFINE VARIABLE dTotalSalesAmount AS DECIMAL NO-UNDO.
DEFINE VARIABLE iColumn AS INTEGER NO-UNDO INITIAL 1.
DEFINE VARIABLE cColumn AS CHARACTER NO-UNDO.
DEFINE VARIABLE cRange AS CHARACTER NO-UNDO.

/* create a new Excel Application object */
CREATE "Excel.Application" chExcelApplication.

/* Launch Excel so it is visible to the user */
chExcelApplication:Visible = TRUE.

/* create a new Workbook */
chWorkbook = chExcelApplication:Workbooks:Add().

/* get the active Worksheet */
chWorksheet = chExcelApplication:Sheets:Item(1).

/* set the column names for the Worksheet */
chWorksheet:Columns("A"):ColumnWidth = 18.
chWorksheet:Columns("B"):ColumnWidth = 12.
chWorksheet:Columns("C"):ColumnWidth = 12.
chWorksheet:Range("A1:C1"):Font:Bold = TRUE.
chWorksheet:Range("A1"):Value = "SalesRep".
chWorksheet:Range("B1"):Value = "Total Sales".
chWorksheet:Range("C1"):Value = "Annual Quota".
```
/* Iterate through the SalesRep table, populate the Worksheet appropriately */
FOR EACH SalesRep NO-LOCK:
    ASSIGN
dAnnualQuota = 0
iTotalNumberOfOrders = 0
dTotalSalesAmount = 0
iColumn = iColumn + 1.

FOR EACH Order OF SalesRep NO-LOCK:
    iTotalNumberOfOrders = iTotalNumberOfOrders + 1.
    FIND Invoice WHERE Invoice.OrderNum = Order.OrderNum NO-ERROR.
    IF AVAILABLE Invoice THEN
        dTotalSalesAmount = dTotalSalesAmount + Invoice.Amount.
    END.
END.

DO iMonth = 1 TO 12:
    dAnnualQuota = dAnnualQuota + Salesep.MonthQuota[iMonth].
END.

ASSIGN
cColumn = STRING(iColumn)
cRange = "A" + cColumn
chWorkSheet:Range(cRange):Value = SalesRep.RepName
cRange = "B" + cColumn
chWorkSheet:Range(cRange):Value = dTotalSalesAmount
cRange = "C" + cColumn
cRange = "D" + cColumn
chWorkSheet:Range(cRange):Value = dAnnualQuota.
END.

chWorkSheet:Range("B2:C10"):Select().
chExcelApplication:Selection:Style = "Currency".

/* create embedded chart using the data in the Worksheet */
chWorksheetRange = chWorksheet:Range("A1:C10").
chWorksheet:ChartObjects:Add(10,150,425,300):Activate.
chExcelApplication:ActiveChart:ChartWizard(chWorksheetRange, 3, 1, 2, 1, 1, TRUE, "1996 Sales Figures", "Sales Person", "Annual Sales").

/* create chart using the data in the Worksheet */
chChart = chExcelApplication:Charts:Add().
chChart:Name = "Test Chart".
chChart:Type = 11.

/* release com-handles */
RELEASE OBJECT chExcelApplication.
RELEASE OBJECT chWorkbook.
RELEASE OBJECT chWorksheet.
RELEASE OBJECT chChart.
RELEASE OBJECT chWorksheetRange.
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.

This chapter describes 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. For information on using the AppBuilder to work with ActiveX controls, see OpenEdge Development: AppBuilder. The information in this chapter relies on ABL support for COM objects. For information on COM objects in ABL, see Chapter 14, “Using COM Objects in ABL.”

This chapter contains the following sections:

- 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 Chapter 8, “Introduction to External Program Interfaces.”

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

Figure 16–1 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 the “ActiveX controls and examples installed with ABL” section on page 16–35.

Figure 16–1: 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). The “Handling events” section on page 16–19 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 Chapter 8, “Introduction to External Program Interfaces.”)

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 16–1 has a location and tab order that is determined by the control-frame widget location and tab order in the Customer Information 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 Chapter 14, “Using COM Objects in ABL.”

Figure 16–2 shows the relationship between the ActiveX control, its control-frame, and other widgets in the ABL user interface.

Figure 16–2: 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 Table 16–1.

Table 16–1: Control-frame attributes and properties

<table>
<thead>
<tr>
<th>Widget attribute</th>
<th>COM object property</th>
</tr>
</thead>
<tbody>
<tr>
<td>HEIGHT-PIXELS, HEIGHT-CHARS</td>
<td>Height</td>
</tr>
<tr>
<td>NAME</td>
<td>Name</td>
</tr>
<tr>
<td>WIDTH-PIXELS, WIDTH-CHARS</td>
<td>Width</td>
</tr>
<tr>
<td>X, COLUMN</td>
<td>Left</td>
</tr>
<tr>
<td>Y, ROW</td>
<td>Top</td>
</tr>
</tbody>
</table>

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 OpenEdge Development: 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.
How ABL supports ActiveX controls

Properties

Table 16–2 lists the available extended properties.

Table 16–2: Extended properties

<table>
<thead>
<tr>
<th>Property name</th>
<th>Type</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>HonorProKeys</td>
<td>logical</td>
<td>read/write at design time</td>
</tr>
<tr>
<td></td>
<td></td>
<td>read/write at runtime</td>
</tr>
<tr>
<td>Name</td>
<td>string</td>
<td>read/write at design time</td>
</tr>
<tr>
<td></td>
<td></td>
<td>read/write at runtime</td>
</tr>
<tr>
<td>Parent</td>
<td>COM-handle</td>
<td>read at runtime</td>
</tr>
<tr>
<td>Tag</td>
<td>string</td>
<td>read/write at design time</td>
</tr>
<tr>
<td></td>
<td></td>
<td>read/write at runtime</td>
</tr>
<tr>
<td>Visible</td>
<td>logical</td>
<td>read/write at design time</td>
</tr>
<tr>
<td></td>
<td></td>
<td>read/write at runtime</td>
</tr>
</tbody>
</table>

Table 16–3 defines each property and how it can be used.

Table 16–3: Property definitions

<table>
<thead>
<tr>
<th>Property</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>HonorProKeys</td>
<td>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.</td>
</tr>
<tr>
<td>Name</td>
<td>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.</td>
</tr>
<tr>
<td>Parent</td>
<td>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.</td>
</tr>
</tbody>
</table>
### 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.

---

**Table 16–3: Property definitions**

<table>
<thead>
<tr>
<th>Property</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>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.</td>
</tr>
<tr>
<td>Visible</td>
<td>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 overridden 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.</td>
</tr>
</tbody>
</table>
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. For complete information on using the AppBuilder to add an ActiveX control, see OpenEdge Development: AppBuilder.

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 the “Accessing ActiveX controls at runtime” section on page 16–12. For more information on locating and sizing the control in the user interface, see the “Orienting a control in the user interface at design time” section on page 16–10.

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 Chapter 14, “Using COM Objects in ABL”), 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 the “Programming ActiveX controls in the AppBuilder” section on page 16–25.

---

### 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 the “Handling events” section on page 16–19.

---

**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 the “Handling events” section on page 16–19.)

**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 the “Managing ActiveX controls at runtime” section on page 16–16.

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.

Figure 16–3 describes the instantiation process for a single ActiveX control.

![Diagram of ActiveX control instantiation](image)

**Figure 16–3:** Instantiating an ActiveX control at runtime

- 1) Create and parent the control-frame widget to a frame.
- 2) Realize the control-frame.
- 3) Load the control.
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)

Figure 16–4 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 16–4:** 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 chF: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 the “Programming ActiveX controls in the AppBuilder” section on page 16–25.

**Using the Control-Name property of the control container**

This example shows how you might access the component handle to the CSSpin control using the Control-Name property:

```plaintext
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:

```plaintext
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 16–4). 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 Chapter 14, “Using COM Objects in ABL.”
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. (For more information, see *OpenEdge Development: AppBuilder*.)

**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 the “Managing external procedure files” section on page 16–24.

**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 Table 16–4.

**Table 16–4: Data type mappings for OCX event parameters**

<table>
<thead>
<tr>
<th>COM data type</th>
<th>ABL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array</td>
<td>ABL array variable</td>
</tr>
<tr>
<td>Array of bytes</td>
<td>RAW</td>
</tr>
<tr>
<td>Boolean</td>
<td>LOGICAL</td>
</tr>
<tr>
<td>Byte</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Currency</td>
<td>DECIMAL</td>
</tr>
</tbody>
</table>
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 Chapter 14, “Using COM Objects in ABL.”

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 placeholder. 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, then see Appendix A, “COM Object Data Type Mapping.” For information on Type Libraries and ABL, see Chapter 14, “Using COM Objects in ABL.”

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.

---

<table>
<thead>
<tr>
<th>COM data type</th>
<th>ABL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Date</td>
<td>DATE</td>
</tr>
<tr>
<td>Double</td>
<td>DATETIME</td>
</tr>
<tr>
<td>Double</td>
<td>DATETIME-TZ</td>
</tr>
<tr>
<td>Double</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>Integer (2-byte integer)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Long (4-byte integer)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Object (COM)</td>
<td>COM-HANDLE</td>
</tr>
<tr>
<td>Single (float)</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>String</td>
<td>CHARACTER</td>
</tr>
<tr>
<td></td>
<td>LONGCHAR</td>
</tr>
<tr>
<td>Unsigned byte</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Unsigned long (4-byte integer)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Unsigned short (2-byte integer)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Variant</td>
<td>ANYTYPE1</td>
</tr>
<tr>
<td>VT-ARRAY (if single-dimensional array of bytes)</td>
<td>RAW</td>
</tr>
</tbody>
</table>

1. You must replace ANYTYPE, provided by the AppBuilder in some OCX event trigger templates, with a valid ABL data type.
Handling events

**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:** If an event procedure has any INPUT-OUTPUT or OUTPUT parameters and contains any I/O-blocking statements, such as SET or UPDATE, the output parameters are not returned to the control.

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.

**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.MyComboBoxNewItem:
    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):

```plaintext
PROCEDURE CtrlFrame-2.MyComboNewItem:
    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 the “Handling ActiveX control events” section on page 16–19), 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 16–1. 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:

```c
/* 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:

```c
/* ******************* 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:

```c
/* ***********************  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.

**Notes:** 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:

```plaintext
/* **********************  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:

```plaintext
/* **************************** 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:

```plaintext
/* ******************************************* 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 the “Initializing the control” section on page 16–31). 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.
```

**Notes:** 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:

```plaintext
PROCEDURE initialize-controls:

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 the “Handling events” section on page 16–19.) 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:

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:

```c
/* ******************************** 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:

```plaintext
/* ********************** Internal Procedures ********************** */

PROCEDURE displayCustomer :

/*---------------------------------------------------------------*/
Purpose:  
Parameters: <none>  
Notes: 
---------------------------------------------------------------*/

iRecordCount = chCSSpin:Value.

    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 you 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.

Example applications using ActiveX controls

In addition to the on-line example, i-ocx1.w, OpenEdge comes installed with a number of sample applications using ActiveX controls that you can use to test and borrow code for your own application development. These applications reside in separate subdirectories under %DLC%\src\samples\ActiveX. Each subdirectory contains a set of files for one application. These files include a readme.txt file that describes the requirements for running the application and the capabilities that it demonstrates.
ABL (Advanced Business Language) provides direct access to TCP/IP sockets, both unsecured sockets and secured sockets using the Secure Sockets Layer (SSL). 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.

This chapter describes how you can access TCP/IP sockets directly from ABL. It assumes that you are familiar with basic TCP/IP socket programming and SSL.

**Note:** SSL incurs heavy performance penalties, depending on the client, server, and network resources and load. For more information, see *OpenEdge Getting Started: Core Business Services*.

This chapter contains the following sections:

- 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
Table 17–1 lists the ABL elements that are either valid only for working with sockets or have special application in socket programming. The remaining sections in this chapter explain how to use these elements.

### Table 17–1: ABL for programming sockets

<table>
<thead>
<tr>
<th>ABL element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTES-READ</td>
<td>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.</td>
</tr>
<tr>
<td>BYTES-WRITTEN</td>
<td>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.</td>
</tr>
<tr>
<td>CONNECT</td>
<td>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 I/O-blocking or PROCESS EVENTS statement, executes any CONNECT event procedure defined for the server socket object. You can use this event procedure to obtain the socket object with which the client is communicating.</td>
</tr>
<tr>
<td>CONNECT( connection-parameters )</td>
<td>A method on the socket object handle that connects a socket handle to a specified TCP/IP port on a specified host.</td>
</tr>
<tr>
<td>CONNECTED( )</td>
<td>A method on the socket object handle that indicates if a socket handle is currently connected to a port.</td>
</tr>
<tr>
<td>CREATE SERVER-SOCKET server-socket-handle</td>
<td>A statement that creates a server socket object with all attributes set to their default values, and stores its handle in a HANDLE variable.</td>
</tr>
<tr>
<td>CREATE SOCKET socket-handle</td>
<td>A statement that creates a socket object with all attributes set to their default values and stores its handle in a HANDLE variable.</td>
</tr>
<tr>
<td>DEFINE INPUT PARAMETER socket-handle</td>
<td>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.</td>
</tr>
</tbody>
</table>
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.

A method on the server socket object handle that indicates that new connections are no longer accepted on the server socket.

A method on the socket object handle that terminates the connection between the socket object and the port to which it is connected.

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.

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.

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.

A method on the socket object handle that indicates the number of bytes available for reading from the socket.

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 OpenEdge Development: ABL Reference and your TCP documentation.

<table>
<thead>
<tr>
<th>ABL element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELETE OBJECT <em>handle</em></td>
<td>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.</td>
</tr>
<tr>
<td>DISABLE-CONNECTIONS( )</td>
<td>A method on the server socket object handle that indicates that new connections are no longer accepted on the server socket.</td>
</tr>
<tr>
<td>DISCONNECT( )</td>
<td>A method on the socket object handle that terminates the connection between the socket object and the port to which it is connected.</td>
</tr>
<tr>
<td>ENABLE-CONNECTIONS (connection-parameters)</td>
<td>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.</td>
</tr>
<tr>
<td>FIRST-SERVER-SOCKET</td>
<td>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.</td>
</tr>
<tr>
<td>FIRST-SOCKET</td>
<td>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.</td>
</tr>
<tr>
<td>GET-BYTES-AVAILABLE( )</td>
<td>A method on the socket object handle that indicates the number of bytes available for reading from the socket.</td>
</tr>
<tr>
<td>GET-SOCKET-OPTION (option-name)</td>
<td>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 OpenEdge Development: ABL Reference and your TCP documentation.</td>
</tr>
</tbody>
</table>
Table 17–1:  ABL for programming sockets  

<table>
<thead>
<tr>
<th>ABL element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAST-SERVER-SOCKET</td>
<td>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.</td>
</tr>
<tr>
<td>LAST-SOCKET</td>
<td>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.</td>
</tr>
<tr>
<td>LOCAL-HOST</td>
<td>A CHARACTER attribute on the socket object handle that returns the IP address on the local machine where the socket object is connected.</td>
</tr>
<tr>
<td>LOCAL-PORT</td>
<td>An INTEGER attribute on the socket object handle that returns the local port number of the socket object.</td>
</tr>
<tr>
<td>NEXT-SIBLING</td>
<td>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.</td>
</tr>
<tr>
<td>MEMPTR</td>
<td>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 Chapter 8, “Introduction to External Program Interfaces.”</td>
</tr>
<tr>
<td>PREV-SIBLING</td>
<td>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.</td>
</tr>
<tr>
<td>PROCESS EVENTS</td>
<td>A statement that you can use to handle any pending CONNECT or READ-RESPONSE events. You can also use any I/O-blocking statement, such as the WAIT-FOR statement.</td>
</tr>
<tr>
<td>READ(</td>
<td>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.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>memptr-expression</th>
<th>position</th>
<th>bytes-to-read</th>
</tr>
</thead>
<tbody>
<tr>
<td>,</td>
<td>,</td>
<td>, mode</td>
</tr>
</tbody>
</table>
Table 17–1: ABL for programming sockets (4 of 5)

<table>
<thead>
<tr>
<th>ABL element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ-RESPONSE</td>
<td>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 I/O-blocking or PROCESS EVENTS statement, executes any READ-RESPONSE event procedure defined for the socket object. You can use this event procedure if you want to read data from the socket in an event-driven manner.¹</td>
</tr>
<tr>
<td>REMOTE-HOST</td>
<td>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.</td>
</tr>
<tr>
<td>REMOTE-PORT</td>
<td>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.</td>
</tr>
<tr>
<td>SELF</td>
<td>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.</td>
</tr>
<tr>
<td>SENSITIVE</td>
<td>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.</td>
</tr>
<tr>
<td>Server socket object handle</td>
<td>A handle to a server socket object. This object allows you to listen for and accept TCP/IP connections on a given port.</td>
</tr>
<tr>
<td>SET-CONNECT-PROCEDURE(event-internal-procedure [, procedure-context ])</td>
<td>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.</td>
</tr>
<tr>
<td>SET-READ-RESPONSE-PROCEDURE(event-internal-procedure [, procedure-context ])</td>
<td>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.</td>
</tr>
<tr>
<td>Socket object handle</td>
<td>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.</td>
</tr>
</tbody>
</table>
Table 17–1:  ABL for programming sockets

<table>
<thead>
<tr>
<th>ABL element</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SET-SOCKET-OPTION   ( name, arguments )</td>
<td>A method on the socket object handle that sets the specified TCP socket option. ABL supports the following options:</td>
</tr>
<tr>
<td></td>
<td>• TCP-NODELAY</td>
</tr>
<tr>
<td></td>
<td>• SO-LINGER</td>
</tr>
<tr>
<td></td>
<td>• SO-KEEPALIVE</td>
</tr>
<tr>
<td></td>
<td>• SO-REUSEADDR</td>
</tr>
<tr>
<td></td>
<td>• SO-SNDBUF</td>
</tr>
<tr>
<td></td>
<td>• SO-RCVBUF</td>
</tr>
<tr>
<td></td>
<td>• SO-RCVTIMEO</td>
</tr>
<tr>
<td></td>
<td>For more information on these options, see OpenEdge Development: ABL Reference and your TCP documentation.</td>
</tr>
<tr>
<td>SSL-SERVER-NAME</td>
<td>A CHARACTER attribute on the socket object handle that returns the SSL server’s X.500 Subject name field after any SSL session has been established.</td>
</tr>
<tr>
<td>TYPE</td>
<td>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.</td>
</tr>
<tr>
<td>WAIT-FOR ...</td>
<td>A statement that you can use to handle any pending CONNECT or READ-RESPONSE events. You can also use PROCESS EVENTS or any other I/O-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.</td>
</tr>
<tr>
<td>WRITE( memptr-expression , position , bytes-to-write )</td>
<td>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.</td>
</tr>
</tbody>
</table>

1. 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 the “Data detection using the event-driven model” section on page 17–13.
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.

Table 17–2 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 17–2: Summary of socket programming tasks

<table>
<thead>
<tr>
<th>Application function</th>
<th>Tasks</th>
</tr>
</thead>
</table>
| Socket server        | • Create a server socket object and enable it to listen for connections (CONNECT events) on a specified port (required). See the “Implementing an ABL socket server” section on page 17–9.  
• Define a CONNECT event procedure to respond to CONNECT events and return the socket object handle created for each connection (required). See the “Listening and responding to connection requests” section on page 17–9.  
• Make the server socket temporarily insensitive to CONNECT events. See the “Managing the server socket” section on page 17–10.  
• Disable the server socket from listening for new socket connections. See the “Managing the server socket” section on page 17–10.  
• Delete the server socket object. See the “Managing the server socket” section on page 17–10. |
| Socket client         | • Create a socket object to read and write data, and connect it to a port on a socket server (required). See the “Implementing an ABL socket client” section on page 17–11. |
Table 17–2: Summary of socket programming tasks

<table>
<thead>
<tr>
<th>Application function</th>
<th>Tasks</th>
</tr>
</thead>
</table>
| Client or server     | • Define and initialize a MEMPTR variable (required to read or write data on a socket). See the “Defining and initializing MEMPTR variables” section on page 17–12.  
• Test whether data is available to read on a socket. See the “Data detection using the procedural model” section on page 17–12.  
• Define a READ–RESPONSE event procedure to respond to data coming in on a socket connection. See the “Data detection using the event-driven model” section on page 17–13.  
• Read data on a socket and test for the number of bytes read for each read operation. See the “Reading data on a socket” section on page 17–14.  
• Write data on a socket and test for the number of bytes written for each write operation. See the “Writing data on a socket” section on page 17–16.  
• Marshall data into a MEMPTR buffer for writing on a socket and unmarshall data from a MEMPTR buffer after reading on a socket. See the “Marshalling and unmarshalling data for socket I/O” section on page 17–16.  
• Test whether a socket object is connected. See the “Managing sockets and their connections” section on page 17–17.  
• Make a socket temporarily insensitive to READ–RESPONSE events. See the “Managing sockets and their connections” section on page 17–17.  
• Set and get socket options. See the “Managing sockets and their connections” section on page 17–17.  
• Obtain the local and remote host IP addresses and port numbers involved in a socket connection. See the “Managing sockets and their connections” section on page 17–17.  
• Disconnect a socket object from its remote host and port. See the “Managing sockets and their connections” section on page 17–17.  
• Delete socket objects. See the “Managing sockets and their connections” section on page 17–17. |
| Socket security      | • Configure socket servers and clients for SSL. See the “Implementing ABL socket security” section on page 17–18.  
• Enable SSL server connections. See the “Enabling SSL server connections” section on page 17–20.  
• Connect an SSL client to an SSL server. See the “Connecting an SSL client to an SSL server” section on page 17–21.  
• Invoke socket operations. See the “Invoking socket operations for SSL sessions” section on page 17–21. |

**Note:** SSL 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-THREAD 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-CONNECTIONS() method in OpenEdge Development: ABL Reference.

3. Specify a CONNECT event procedure using the SET-CONNECT-PROCEDURE() method.

4. Wait for CONNECT events using a blocking I/O 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, ABL listens on the specified port for client connections. When a socket client sends a connection request to this port, ABL automatically accepts the connection.

After accepting the connection, ABL posts a CONNECT event on the server socket and calls the CONNECT event procedure that you have specified using the SET-CONNECT-PROCEDURE() method. The CONNECT event procedure executes, returning a handle to a socket object created by ABL and passed as a parameter to the event procedure context. This socket object, then, is the communications endpoint for the connection on the server.

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 blocking I/O 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.
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 the “Read, writing, and managing sockets on clients and servers” section on page 17–12.

**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 the “Read, writing, and managing sockets on clients and servers” section on page 17–12.
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 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. 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 Chapter 8, “Introduction to External Program Interfaces.”

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 the “Reading data on a socket” section on page 17–14.
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. You can set up this event handler as follows:

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-CONNECT-PROCEDURE( ) method on the socket.

3. Include blocking I/O 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). 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**

```plaintext
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 Table 17–3.

**Table 17–3: Socket reading modes**

<table>
<thead>
<tr>
<th>Compiler constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ-AVAILABLE</td>
<td>1</td>
<td>The <code>READ( )</code> 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 <code>bytes-to-read</code> bytes.</td>
</tr>
<tr>
<td>READ-EXACT-NUM</td>
<td>2</td>
<td>(Default) The <code>READ( )</code> method blocks until <code>bytes-to-read</code> bytes are available to read from the socket.</td>
</tr>
</tbody>
</table>

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 Table 17–4 illustrates.

**Table 17–4: Effect of read mode and timeout value on `READ()`’s timeout behavior**

<table>
<thead>
<tr>
<th>Read mode</th>
<th>With a timeout value</th>
<th>Without a timeout value</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ-AVAILABLE</td>
<td>Assuming there are no connection failures, <code>READ()</code> blocks until one of the following occurs:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• The timeout expires</td>
<td>Assuming there are no connection failures, <code>READ()</code> blocks until there is at least one byte available to read on the socket</td>
</tr>
<tr>
<td></td>
<td>• There is at least one byte available to read on the socket</td>
<td></td>
</tr>
<tr>
<td>READ-EXACT-NUM</td>
<td>Assuming there are no connection failures:</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• If there is any data on the socket, <code>READ()</code> blocks until there are <code>bytes-to-read</code> bytes available to read on the socket</td>
<td>Assuming there are no connection failures, <code>READ()</code> blocks until there are <code>bytes-to-read</code> bytes available to read on the socket</td>
</tr>
<tr>
<td></td>
<td>• Else, <code>READ()</code> blocks until the timeout expires</td>
<td></td>
</tr>
</tbody>
</table>

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).

The is the syntax of the `WRITE()` method:

**Syntax**

```plaintext
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 Chapter 8, “Introduction to External Program Interfaces.”
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.
Implementing ABL socket security

You can use the Secure Sockets Layer (SSL) to provide a security infrastructure that protects communications between a socket client and server. SSL 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 (SSL client) and the responder (SSL server). The server is identified by the private key that it stores and the client is identified as a valid SSL client for that server by the public key that it stores and provides to the server. SSL 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 SSL server.

Like socket communications in general, SSL is both application and transport independent. This section describes how to implement and manage SSL for ABL applications communicating over TCP/IP sockets.

For more information on SSL and how it uses private and public keys and public key certificates to handle security tasks in these contexts, see OpenEdge Getting Started: Core Business Services.

Note: SSL 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 SSL:

1. On your ABL socket server, create and configure the private key and certificate store required to identify your socket server as an SSL server.
2. On your ABL socket client, create and configure the public key certificate store required to access your SSL server as an SSL client.
3. Startup and ensure that your ABL socket server has enabled connections for SSL.
4. Startup and connect your ABL client as an SSL client to your SSL server.
5. Invoke socket operations in SSL sessions.

The sections that follow describe these tasks.
Configuring SSL 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 SSL. For all OpenEdge clients and servers, OpenEdge also installs built-in key and certificate stores that provide a simple, default SSL implementation for applications that require a minimal level of SSL support. SSL servers and clients have different SSL configuration requirements. So, if your SSL server is also an SSL client of some other SSL server, you must configure your application for both.

Note: Be very sure you need SSL before using this option. SSL incurs more or less heavy performance penalties, depending on resources and load.

Configuring SSL servers

For an SSL server, you must have installed a private key and digital (public key) certificate that uniquely identifies your ABL socket server as an SSL server and allows all communications to be encrypted between it and any SSL 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 thereby eliminates the need for client-server authentication to complete SSL connections.

However, to create a complete SSL implementation that supports all the features of SSL, you must obtain a unique private key and server digital certificate from an industry-recognized certificate authority (a CA such as Verisign, RSA, or Thawte) 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 SSL connections.

For more information on OpenEdge SSL support, CA’s, keys, digital certificates, and using pkiutil, see OpenEdge Getting Started: Core Business Services.

Configuring SSL clients

For an SSL client, you must have installed a public key certificate that allows the client to authenticate and encrypt communications with a specific SSL server that it connects to. You can obtain the required public key certificate for a given SSL 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 SSL connection to an SSL server.

For more information on OpenEdge SSL support, public key certificates, and using certutil, see OpenEdge Getting Started: Core Business Services.
Enabling SSL server connections

When you enable SSL connections on an ABL server socket object, the socket accepts connections only from SSL clients that can authenticate to your ABL application as an SSL server.

**Note:** Be very sure you need SSL before using this option. SSL incurs more or less heavy performance penalties, depending on resources and load.

To enable SSL connections on an ABL server socket object, include the SSL parameters shown in Table 17–5 as part of the connection parameters string that you pass to the server socket object ENABLE-CONNECTIONS() method. For additional information, see the “Implementing an ABL socket server” section on page 17–9.

Table 17–5: Server socket object SSL connection parameters

<table>
<thead>
<tr>
<th>Connection parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-ssl</td>
<td>Specifies the all connections to this server socket must use SSL.</td>
</tr>
<tr>
<td>-keyalias aliasname</td>
<td>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.</td>
</tr>
<tr>
<td>-keyalaispasswd encrypted-password</td>
<td>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.</td>
</tr>
<tr>
<td>-nosessioncache</td>
<td>If specified, caching for the SSL client session is disabled.</td>
</tr>
<tr>
<td>-sessiontimeout [seconds]</td>
<td>Specifies, in seconds, the length of time that an SSL client session is held in the session cache, during which an SSL client can resume its session. The default is 180 seconds.</td>
</tr>
</tbody>
</table>

For more information on OpenEdge SSL support, key and certificate stores, the genpassword utility, and enabling connections to SSL servers, see *OpenEdge Getting Started: Core Business Services*. 
Connecting an SSL client to an SSL server

When you connect an ABL socket object to an SSL server socket, you must specify the connection as an SSL connection. Your connecting client must be able to authenticate itself to the SSL server, and you can optionally require the server to authenticate itself to your SSL client.

**Note:** Be very sure you need SSL before using this option. SSL incurs more or less heavy performance penalties, depending on resources and load.

To connect an ABL socket object to an SSL server socket, include the SSL parameters shown in Table 17–6 as part of the connection parameters string that you pass to the socket object `CONNECT()` method. For additional information, see the “Implementing an ABL socket client” section on page 17–11.

<table>
<thead>
<tr>
<th>Table 17–6: Client socket object SSL connection parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Connection parameter</strong></td>
</tr>
<tr>
<td>-ssl</td>
</tr>
<tr>
<td>-nosessionreuse</td>
</tr>
<tr>
<td>-nohostverify</td>
</tr>
</tbody>
</table>

For more information on OpenEdge SSL support and connecting an SSL client to an SSL server, see *OpenEdge Getting Started: Core Business Services*.

Invoking socket operations for SSL sessions

For working in an SSL 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 SSL.

**Identifying the SSL server in a client session**

After connecting to an SSL server socket from a client socket object using SSL, 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-SSL client socket connects to an SSL server socket

When you open a non-SSL connection from an ABL client to an ABL SSL server, you can initiate a situation that can result in a deadlock condition that is unbreakable from within the application. This condition results from the following sequence of events:

1. The non-SSL 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 SSL 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-SSL 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 SSL 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 SSL 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 (i-sktsv1.p) and a socket client (i-sktcl1.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.

```
i-sktsv1.p
(1 of 2)

DEFINE VARIABLE aOk AS LOGICAL NO-UNDO.
DEFINE VARIABLE cString AS CHARACTER NO-UNDO.
DEFINE VARIABLE greeting AS CHARACTER NO-UNDO.
DEFINE VARIABLE hServerSocket AS HANDLE NO-UNDO.
DEFINE VARIABLE hSocket AS HANDLE NO-UNDO.
DEFINE VARIABLE mBuffer AS MEMPTR NO-UNDO.

MESSAGE "We are in the socket server".
CREATE SERVER-SOCKET hServerSocket.

/* Marshal data to write */
ASSIGN
   SET-SIZE(mBuffer) = 64
   greeting = "SERVER - hello"
   PUT-STRING(mBuffer,1) = greeting.

MESSAGE "Start event handling".
hServerSocket:SET-CONNECT-PROCEDURE("connProc").
aOk = hServerSocket:ENABLE-CONNECTIONS("-S 3333").
MESSAGE "Enabled connections:" aOk.
IF NOT aOk THEN RETURN.

/* This WAIT-FOR completes after the first connection */
WAIT-FOR CONNECT OF hServerSocket.

MESSAGE "Freeing up resources".
hServerSocket:DISABLE-CONNECTIONS().
DELETE OBJECT hServerSocket.
MESSAGE "Finished".

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 CONNECT event procedure, connProc".
      hSocket:WRITE(mBuffer,1,LENGTH(greeting)).
      MESSAGE hSocket:BYTES-WRITTEN "bytes written".
      hSocket:DISCONNECT().
      DELETE OBJECT hSocket.
      END.
   ELSE MESSAGE "Unable to obtain socket.".
   END PROCEDURE.
```
/* Do READ-RESPONSE event handling */
MESSAGE "Start event handling".
hSocket:SET-READ-RESPONSE-PROCEDURE("readProc").

/* This WAIT-FOR completes after the first data reception */
WAIT-FOR READ-RESPONSE OF hSocket.

MESSAGE "Freeing up resources".
hSocket:DISCONNECT().
DELETE OBJECT hSocket.
MESSAGE "Finished".

PROCEDURE readProc:
/* Read procedure for socket */
DEFINE VARIABLE mBuffer AS MEMPTR NO-UNDO.

MESSAGE "We are in READ-RESPONSE event procedure, readProc".
SET-SIZE(mBuffer) = 64.
SELF:READ(mBuffer,1,SELF:GET-BYTES-AVAILABLE()).
MESSAGE SELF:BYTES-READ "bytes read".
cString = GET-STRING(mBuffer,1). /* Unmarshal data */
DISPLAY cString FORMAT "x(64)".
END PROCEDURE.

i-sktsv1.p

/* Do READ-RESPONSE event handling */
MESSAGE "Start event handling".
hSocket:SET-READ-RESPONSE-PROCEDURE( "readProc").

/* This WAIT-FOR completes after the first data reception */
WAIT-FOR READ-RESPONSE OF hSocket.

MESSAGE "Freeing up resources".
hSocket:DISCONNECT().
DELETE OBJECT hSocket.
MESSAGE "Finished".

i-sktscl1.p
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.

### i-sktsv2.p

```abl
PROCEDURE readProc:
   /* Read procedure for socket */
   DEFINE VARIABLE mBuffer AS MEMPTR NO-UNDO.
   MESSAGE "We are in READ-RESPONSE event procedure, readProc".
   SET-SIZE(mBuffer) = 64.
   SELF:READ(mBuffer,1,SELF:GET-BYTES-AVAILABLE()).
   MESSAGE SELF:BYTES-READ "bytes read".
   cString = GET-STRING(mBuffer,1). /* Unmarshal data */
   DISPLAY cString FORMAT "x(64)".
END PROCEDURE.
```

/* 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.

/* Put customer record into a RAW variable and store in local DB. */
RAW-TRANSFER customer TO r1.
RELEASE customer.
```

### i-sktcl2.p

```abl
/* 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.

/* Put customer record into a RAW variable and store in local DB. */
RAW-TRANSFER customer TO r1.
RELEASE customer.
```

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.
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.
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-CONNECT-PROCEDURE( "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**

```plaintext
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.
hssocket:DISCONNECT().
DELETE OBJECT hsocket.
```
Host Language Call Interface

This chapter describes the OpenEdge® Host Language Call (HLC) Interface in the following sections:

- HLC and OpenEdge
- Using HLC
- Overview of HLC
- HLC files and directories
- Writing C functions
- Avoiding common HLC errors
- Memory allocation
- Data size
- Using HLC library functions
- Building an HLC executable
- HLC applications on UNIX systems
- Compiling C source files
- Example HLC application
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 Chapter 12, “Shared Library and DLL Support”).

Figure 18–1 shows a top-down structure diagram for an OpenEdge application that calls your application functions using HLC.

---

**Figure 18–1: OpenEdge application calling functions with HLC**

**Requirements for using HLC**

To use HLC, you must be proficient with the C language and the C language compiler and linker on your system.
Using 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 Chapter 12, “Shared Library and DLL Support.” 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 the “Using HLC library functions” section on page 18–16 section.
Overview of HLC

You can use HLC to add virtually any feature to ABL that is written in C and that follows HLC programming rules. To use HLC, you should be familiar with the following topics:

- Using the ABL client.
- Using the OEBuild utility. For more information on the OEBuild Utility, see the appendix on building executables in *OpenEdge Deployment: Managing ABL Applications*.
- Designing, compiling, and linking C programs. See the C documentation for more information.

Figure 18–2 illustrates the steps to build an HLC executable.

![Building an HLC executable](image)

**Figure 18–2: Steps to build an HLC executable**

1. Write your C functions and update the PRODSP() dispatch routine contained in hlprodsp.c.
2. Compile your C routine source files and the hlprodsp.c dispatch routine source file.
3. Use the OEBuild utility to generate a link script for your executable.
4. Link your C object files, the hlprodsp.c dispatch routine object file, and ABL object files, using the link script that OEBuild generates.
For more information on the OEBuild utility, see the appendix on building executables in *OpenEdge Deployment: Managing ABL Applications*.

To use HLC you must build an HLC executable. An HLC executable is an ABL module with your C functions linked. Once you build an HLC executable, you can use the **CALL** statement to execute a linked C function from an ABL procedure.

### Using 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.

Figure 18–3 demonstrates how to run the **CALL** statement.

---

**Figure 18–3: 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 the “Using a library function that writes to a shared buffer” section on page 18–20.

The CALL statement transfers control to the HLC dispatch routine, PRODSP(), passing it your routine identifier and any arguments. The example in Figure 18–3 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 Figure 18–3 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.

**Mapping 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 UNIX 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**

```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. */

FUNCTEST("HLCROUTINE1", hlcfunc1);
FUNCTEST("HLCROUTINE2", hlcfunc2);
```

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():

```c
FUNCTEST("HLCROUTINE1", hlcfunc1);
FUNCTEST("HLCROUTINE2", hlcfunc2);
```

**Note:** The CALL statement and FUNCTEST declaration must use the same letter case for the routine identifier.

This is the syntax for the FUNCTEST macro in hlprodsp.c:

**Syntax**

```c
FUNCTEST ( "routine-identifier", function-name );
```

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:

```c
FUNCTEST("hlcfunc", hlcfunc);
```

When you compile `hlprodsp.c`, the C compiler translates the `FUNCTEST` macro references to C code, for example:

```c
FUNCTEST("HLCROUTINE", hlcfunc);
```

Translates to:

```c
if (strcmp("HLCROUTINE", pfunnam) == 0)  
    return hlcfunc(argc, argv);
```

Therefore, when ABL invokes `PRODSP()` with the argument `HLCROUTINE`, it runs `hlcfunc`.
HLC files and directories

The default HLC installation process creates the directories listed in Table 18–1.

**Table 18–1: HLC directories**

<table>
<thead>
<tr>
<th>UNIX</th>
<th>Windows</th>
</tr>
</thead>
<tbody>
<tr>
<td>$DLC/oebuild/hlc</td>
<td>%DLC\oebuild\hlc</td>
</tr>
<tr>
<td>$DLC/oebuild/hlc/examples</td>
<td>%DLC\oebuild\hlc\examples</td>
</tr>
</tbody>
</table>

The directories in Table 18–2 contain the files listed in Table 18–2.

**Table 18–2: HLC filenames**

<table>
<thead>
<tr>
<th>Filename</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>c</td>
<td>Script to compile your C functions with the correct parameters to use with HLC</td>
</tr>
<tr>
<td>hlc.h</td>
<td>HLC function library header file</td>
</tr>
<tr>
<td>hlprodsp.c</td>
<td>Prototype HLC dispatch routine file</td>
</tr>
</tbody>
</table>

The examples directory contains HLC example files for an oil tank demo application called Tank, described later in this document.
**Writing C functions**

Figure 18–4 shows the relationship between the `PRODSP()` dispatch routine that ABL supplies 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 h1prodsp. c file, your top-level function must use the following syntax:

Syntax

```c
long function-name ( argc, 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 the “Mapping routine identifiers using PRODSP()” section on page 18–6.

Returning 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:

```c
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.
**Naming 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.
Avoiding 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 the “Returning error codes from a top-level function” section on page 18–11 section 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.
Using 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

Accessing 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:

```c
#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 Appendix C, “HLC Library Function Reference.”

This is the syntax for the **prordbi()** function:

**Syntax**

```c
int prordbi ( pbufnam, fhandle, index, pvar, punknown )
    char *pbufnam ;
    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.
Figure 18–5 shows a call being made to `prordbi()` that illustrates HLC programming guidelines.

```
long hlcfunc(argc,argv)                /* Sample HLC function */
int     argc;
char   *argv[];
{
    int    fhandle;        /* field handle */
    long   field_val;      /* variable to store field value */
    int    unk_flag;       /* variable to store ? flag value */
    ...
    fhandle = profldix("hlcbuff","cust-num");
    prordbi("hlcbuff",fhandle,0,&field_val,&unk_flag);
    ...
}
```

**Guideline 1:** Use correct parameter data types.

**Guideline 2:** Use correct parameter values; in this example, HLC library function `profldix()`.

**Guideline 3:** Use correct pointers. The addresses are passed, the variables themselves are not.

---

**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.

**Converting 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.

**Converting 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 Appendix C, “HLC Library Function Reference” for details on the parameters each HLC library function uses and the data types of these parameters.

**Calling 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:

```c
#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 Appendix C, “HLC Library Function Reference” 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 Figure 18–6.

```c
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 */}
```

**Figure 18–6:** Timer-service functions on UNIX
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.

Using 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, prowthebe()), 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.
Passing 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.
Building an HLC executable

After designing the CALL statements, supporting C functions, and the PRODSP() dispatch routine for your application, you can build an HLC executable.

To build an HLC executable:

1. On UNIX operating systems, set up your environment using the ABL BUILDENV utility. This command sets the search paths and options required with your compiler and linker to build ABL executables.

2. Compile your copy of the hlprodsp.c source file that contains the HLC dispatch routine and the source files that contain your C functions. For more information, see the “Compiling C source files” section on page 18–26 section.

3. Use the OEBuild utility to generate a link script for your executable. The OEBuild utility allows you to do the following:
   - Select the OpenEdge product and configurable elements (including HLC) that you want to build into your executable.
   - Enter the filenames of your object files and hlprodsp.o in the dialog box that OEBuild displays. The OEBuild utility inserts the filenames in the link script it generates.

4. Link your executable with the link script that OEBuild generates. This step produces your HLC executable.

For more information on building OpenEdge executables, see the appendix on building executables in *OpenEdge Deployment: Managing ABL Applications*. 
HLC applications on UNIX systems

This section explains how to handle special features of UNIX operating systems for HLC applications, including:

- Handling raw disk I/O
- Handling terminal I/O
- Handling abnormal exits

Handling 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

Handling 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.

Using 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()`:

```abl
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:

```abl
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 *OpenEdge Development: 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.
Handling 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:

```c
/*
 * 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
   ```
Compiling C source files

Compile the hlprodsp.c dispatch routine source file and your C routine source files. For example, the following command compiles source files source.c and hlprodsp.c:

```
c source.c hlprodsp.c
```

The c script is in $DLC/oebuil/hlc on UNIX and %DLC%oebuil\hlc in Windows. It contains a number of options you might want to modify for your particular application environment.
Example HLC application

This section explains how to use the OpenEdge HLC tank demo application. This application is a prepackaged application that you can use to verify that your HLC environment is set up correctly. The C code and sample output for this example are provided in the $DLC/oebuild/hlc/examples directory on UNIX, and in the %DLC%/oebuild/hlc/examples directory in Windows.

Set up your environment and compile this example before you try to run it to verify that your compiler and the OEBuild utility are working correctly. This also helps you learn how to use HLC in a controlled environment.

This sample application uses OpenEdge to keep track of oil storage tanks. Your ABL procedure calls a C program, AVCALC, to calculate the available capacity for a given tank. Figure 18–7 shows that the tanks are cylindrical, with their axes parallel to the level ground.

![Figure 18–7: Tank positioning and orientation](image)

To calculate the available capacity (empty portion) of the tank, you need to know the tank’s diameter, length, and current level of oil. Use the variables in the following formula to calculate the available tank volume:

\[
\begin{align*}
\text{length} \cdot r \cdot \left[ \frac{p}{2} + \sqrt{1 - \left(1 - \frac{\text{level}}{r}\right)^2 \left(1 - \frac{\text{level}}{r}\right)} + \sin^{-1} \left(1 - \frac{\text{level}}{r}\right) \right]
\end{align*}
\]

\[r\]
Radius of the tank.

\[\text{length}\]
Length of the tank.

\[\text{level}\]
Level of oil in the tank.

For this example, assume there is a tank table for this application that contains the following decimal fields:

\[\text{radius}\]
Radius of tank.

\[\text{tlength}\]
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.

Figure 18–8 shows the Data Dictionary report for the tank 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:  

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
<th>Ext</th>
<th>Dec</th>
<th>Format</th>
<th>Init</th>
</tr>
</thead>
<tbody>
<tr>
<td>* tank-id</td>
<td>char</td>
<td></td>
<td></td>
<td>x(8)</td>
<td></td>
</tr>
<tr>
<td>radius</td>
<td>dec</td>
<td>2</td>
<td>&gt;&gt;</td>
<td>9.99</td>
<td>0</td>
</tr>
<tr>
<td>tlength</td>
<td>dec</td>
<td>2</td>
<td>&gt;&gt;</td>
<td>9.99</td>
<td>0</td>
</tr>
<tr>
<td>depth</td>
<td>dec</td>
<td>2</td>
<td>&gt;&gt;</td>
<td>9.99</td>
<td>0</td>
</tr>
<tr>
<td>tavail</td>
<td>dec</td>
<td>2</td>
<td>&gt;&gt;</td>
<td>9.99</td>
<td></td>
</tr>
</tbody>
</table>

Index Name Unique Field Name Seq Ascending abbreviate
------------- -------- --------------- --- --------- ----------
# tank         yes    tank-id        1 yes       yes

Field Validation Criteria, Validation Messages

<table>
<thead>
<tr>
<th>Field</th>
<th>Criterion</th>
<th>Validation Message</th>
</tr>
</thead>
<tbody>
<tr>
<td>depth</td>
<td>depth le 2 * (input radius)</td>
<td>Depth cannot exceed diameter of tank</td>
</tr>
</tbody>
</table>

Help Messages

tank-id : Tank identification number
radius : Radius of oil tank
length : 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). Figure 18–9 shows the ABL procedure that invokes the C function by calling the HLC routine, AVCALC.

```c
/* 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.
```

**Figure 18–9:** ABL procedure calling HLC routine AVCALC

Make a copy of the HLC dispatch routine, hlprodsp.c, and name it tankdsp.c. Modify the routine so that 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**

```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

```c
#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;
        }
```
Example HLC application

```c
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);
```
Running the sample application in Windows

To verify that your HLC application is installed correctly in Windows, run the sample application.

To run the sample application in Windows:

1. Create a working directory, and then go to it.

   **Note**: Before you develop your own applications, move c.bat to a directory that is in your path and move hlc.h to your include file directory or current working directory.

2. Copy the example HLC files to your working directory:

   ```
   copy %DLC%\oebuild\hlc\examples\*.*
   ```

3. Create an empty database:

   ```
   prodb tankdb empty.
   ```

4. Load and run the ABL programs loaddb.p and initdb.p.

5. Compile the C source files to create object files in your current directory:

   ```
   c tankdsp.c hlvcalc.c
   ```
6. Run the OEBuild utility using tankdsp and hlvcalc as object files for the HLC application.

For more information on the OEBuild utility, see the appendix on building executables in OpenEdge Deployment: Managing ABL Applications.

7. Copy the .def and .res files to your current working directory:

```batch
copy %DLC%\oebuild\_prowin.*
```

8. Link your test HLC module using the link script OEBuild generates:

```batch
link @_prowin.lnk
```

9. Link in the resource files:

```batch
rc -k _prowin.res
```


After you run hltank.p, verify that the results you obtain are correct by comparing them to the following results that list the input values for radius, tlength, and depth, and the correct output for tavail:

<table>
<thead>
<tr>
<th>After calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>radius</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>5.00</td>
</tr>
<tr>
<td>5.00</td>
</tr>
<tr>
<td>5.00</td>
</tr>
<tr>
<td>10.00</td>
</tr>
<tr>
<td>10.00</td>
</tr>
</tbody>
</table>
Running the sample application on UNIX

To verify that your HLC application is installed correctly on UNIX, run the sample application.

To run the sample application on UNIX:

1. Run the buildenv script provided in the $DLC/oebuild/eucapp directory to set up your HLC environment and to set environment variables for the C compiler and linker.

2. Create a working directory, then go to it.

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 buildenv to a directory that is in your path and move hlc.h to your include file directory or your current working directory.

4. Enter `testhlc` on 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 that 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:

<table>
<thead>
<tr>
<th>After calculation</th>
</tr>
</thead>
<tbody>
<tr>
<td>radius</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>5.00</td>
</tr>
<tr>
<td>5.00</td>
</tr>
<tr>
<td>5.00</td>
</tr>
<tr>
<td>10.00</td>
</tr>
<tr>
<td>10.00</td>
</tr>
</tbody>
</table>
Here is the `testhlc` script on UNIX:

```
#!/bin/sh
# testhlc -- single-user test of HLC
# 1
DLC=${DLC-'/usr/dlc'}; export DLC
DLCDB=${DLCDB-$DLC}; export DLCDB

#2
SOURCEDB=$DLCDB/empty
DBNAME=hlcdemo
rm -fr $DBNAME.*
prodb $DBNAME $SOURCEDB

#3
echo Now compiling *.c ...
c *.c

#4
echo Now linking test hlc module ...
./ldhlcrx

#5
./_progres $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
else
fi
```

These 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

**Note:** This link script is specific to the sample application. For your own application development, use the OEBuild utility to automatically generate a tailored link script.

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.

```plaintext
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**

```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 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;
}
```
The hldemo.p example is the front end to the sample test application.

**hldemo.p**

```plaintext
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.
```
Appendices

Appendix A, COM Object Data Type Mapping
Appendix B, Audit Policy Maintenance APIs
Appendix C, HLC Library Function Reference
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 Chapter 14, “Using COM Objects in ABL.”

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 Chapter 16, “ActiveX Control Support.”

This appendix contains the following sections:

- Data type conversion strategy
- Conversions from ABL to COM data types
- Conversions from COM to ABL 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 Chapter 14, “Using COM Objects in ABL.”

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 Chapter 14, “Using COM Objects in ABL.” For more information on OCX event procedure templates, see Chapter 16, “ActiveX Control Support.”
Conversions from COM to ABL data types

Table A–1 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 Table A–1, ABL tries to convert it.

Table A–1:  Default conversions from COM data types to ABL data types

<table>
<thead>
<tr>
<th>COM data type</th>
<th>ABL data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolean</td>
<td>LOGICAL</td>
</tr>
<tr>
<td>Signed Byte</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Unsigned Byte</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Signed Short (2-byte integer)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Unsigned Short (2-byte integer)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Signed Integer (4-byte integer)</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Unsigned Integer (4-byte integer)</td>
<td>INT64</td>
</tr>
<tr>
<td>Signed 8-byte Integer</td>
<td>INT64</td>
</tr>
<tr>
<td>Unsigned 8-byte Integer</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>Byte[] (Array)</td>
<td>RAW</td>
</tr>
<tr>
<td>Currency</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>Date</td>
<td>DATETIME</td>
</tr>
<tr>
<td>Double</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>Error</td>
<td>INTEGER</td>
</tr>
<tr>
<td>Float (Single)</td>
<td>DECIMAL</td>
</tr>
<tr>
<td>Object</td>
<td>COM-HANDLE</td>
</tr>
<tr>
<td>String</td>
<td>CHARACTER</td>
</tr>
</tbody>
</table>

Notes: The names for COM data types in Table A–1 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 A–3.

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

Table A–2 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 Table A–2. 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 A–2: Conversions from ABL to COM data types

<table>
<thead>
<tr>
<th>ABL data type</th>
<th>COM data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHARACTER</td>
<td>String</td>
</tr>
<tr>
<td>COM–HANDLE</td>
<td>Object</td>
</tr>
<tr>
<td>DATE</td>
<td>Date</td>
</tr>
<tr>
<td>DATETIME</td>
<td>Date</td>
</tr>
<tr>
<td>DATETIME–TZ</td>
<td>Date</td>
</tr>
<tr>
<td>DECIMAL</td>
<td>Double</td>
</tr>
<tr>
<td>INTEGER</td>
<td>Signed Integer (4 bytes)</td>
</tr>
<tr>
<td>INT64</td>
<td>Signed 8-byte Integer</td>
</tr>
<tr>
<td>LOGICAL</td>
<td>Boolean</td>
</tr>
<tr>
<td>LONGCHAR</td>
<td>String</td>
</tr>
<tr>
<td>RAW</td>
<td>Array of bytes</td>
</tr>
<tr>
<td>anytype</td>
<td>Variant</td>
</tr>
</tbody>
</table>

Notes: 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 Table A–2 does not show a default CHARACTER to Currency conversion pair.

The names for COM data types in Table A–2 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 A–3.
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 A–2.

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:

```abl
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:

```text
[ 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:

```abl
dtXmas = ComObj:Holidays[2].
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 A–1 and Table A–2. However, some documentation and some COM object viewers might use an alternative nomenclature. Table A–3 shows the most common alternates.

Table A–3: Alternate COM data type names

<table>
<thead>
<tr>
<th>Alternative name</th>
<th>Common name</th>
</tr>
</thead>
<tbody>
<tr>
<td>VT_ARRAY</td>
<td>Array</td>
</tr>
<tr>
<td>VT_BOOL</td>
<td>Boolean</td>
</tr>
<tr>
<td>VT_BSTR</td>
<td>String</td>
</tr>
<tr>
<td>VT_BYREF</td>
<td>Pointer</td>
</tr>
<tr>
<td>VT_BYREF + VT_VARIANT</td>
<td>Variant Pointer</td>
</tr>
<tr>
<td>VT_CY</td>
<td>Currency</td>
</tr>
<tr>
<td>VT_DATE</td>
<td>Date</td>
</tr>
<tr>
<td>VT_DISPATCH</td>
<td>Object</td>
</tr>
<tr>
<td>VT_ERROR</td>
<td>Error Code</td>
</tr>
<tr>
<td>VT_I1</td>
<td>Signed Byte</td>
</tr>
<tr>
<td>VT_I2</td>
<td>Signed Short (2-byte integer)</td>
</tr>
<tr>
<td>VT_I4</td>
<td>Signed Integer (4-byte integer)</td>
</tr>
<tr>
<td>VT_I8</td>
<td>Signed 8-byte Integer</td>
</tr>
<tr>
<td>VT_PTR</td>
<td>Pointer</td>
</tr>
<tr>
<td>VT_PTR + VT_VARIANT</td>
<td>Variant Pointer</td>
</tr>
<tr>
<td>VT_R4</td>
<td>Float (Single)</td>
</tr>
<tr>
<td>VT_R8</td>
<td>Double</td>
</tr>
<tr>
<td>VT_UI1</td>
<td>Unsigned Byte</td>
</tr>
<tr>
<td>VT_UI2</td>
<td>Unsigned Short (2-byte integer)</td>
</tr>
<tr>
<td>VT_UI4</td>
<td>Unsigned Integer (4-byte integer)</td>
</tr>
<tr>
<td>VT_UI8</td>
<td>Unsigned 8-byte Integer</td>
</tr>
<tr>
<td>VT_UNKNOWN</td>
<td>Object</td>
</tr>
<tr>
<td>VT_VARIANT</td>
<td>Variant (anytype)</td>
</tr>
</tbody>
</table>
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 *OpenEdge Getting Started: Core Business Services*.

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.

This appendix contains the following sections:

- API overview
- Generic utility API
- Caching API
- Exporting policies to XML
- Importing policies from XML
- Additional audit policy procedures

For basic information on implementing auditing tools in ABL, see Chapter 3, “Auditing.”
API overview

The available APIs for 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

Table B–1 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>
<thead>
<tr>
<th>This procedure file in src\auditing...</th>
<th>Performs this function...</th>
<th>As described in the...</th>
</tr>
</thead>
<tbody>
<tr>
<td>_aud-utils.p</td>
<td>Reads and saves audit configurations using a generic utility API (persistent)</td>
<td>“Generic utility API” section on page B–5</td>
</tr>
<tr>
<td>_aud-cache.p</td>
<td>Reads and saves audit configurations using a caching API (persistent)</td>
<td>“Caching API” section on page B–16</td>
</tr>
<tr>
<td>_get-db-list.p</td>
<td>Gets a list of connected databases with additional information about each database</td>
<td>“Additional audit policy procedures” section on page B–29</td>
</tr>
<tr>
<td>_exp-policies.p</td>
<td>Exports policies from a ProDataSet to an XML file</td>
<td>“Exporting policies to XML” section on page B–24</td>
</tr>
<tr>
<td>_exp-policies-db.p</td>
<td>Exports policies from a connected database to an XML file</td>
<td>“Exporting policies to XML” section on page B–24</td>
</tr>
<tr>
<td>_get-audevents.p</td>
<td>Fills the specified ProDataSet with the _aud-event table records from a specified database</td>
<td>“Additional audit policy procedures” section on page B–29</td>
</tr>
<tr>
<td>_aud-conflict.p</td>
<td>Either checks for audit policy conflicts or aggregates all active audit policies, returning a list of all data being audited</td>
<td>“Additional audit policy procedures” section on page B–29</td>
</tr>
<tr>
<td>_get-policies.p</td>
<td>Reads the audit policies from a specified database into the specified ProDataSet</td>
<td>“Additional audit policy procedures” section on page B–29</td>
</tr>
<tr>
<td>_imp-policies.p</td>
<td>Imports an XML file containing audit policies into either a database or a ProDataSet using an internal API (persistent)</td>
<td>“Importing policies from XML” section on page B–25</td>
</tr>
<tr>
<td>_update-policies.p</td>
<td>Saves to a database the tracked changes to policies stored in a ProDataSet</td>
<td>“Additional audit policy procedures” section on page B–29</td>
</tr>
</tbody>
</table>
Using the generic utility and caching APIs with an AppServer

The generic utility and caching APIs support execution on an AppServer. They provide functions to define the AppServer 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 AppServer whenever they detect that the database is not local. For AppServer execution, you can only call the procedures in the caching or generic utility APIs directly, as they are the only APIs that are AppServer aware.

If you want to call any of the other standalone procedures directly, your code needs to handle the forwarding of requests to the AppServer. For example, in the following combination of pseudo-code and ABL, the caller handles the AppServer connection:

```ABL
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 the policies-dataset-read-xml procedure 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 the override-policies-from-xml procedure to override the policies.

Parameters: None

Example: The following example is a combination of pseudo-code and ABL that shows how the cancel-import-from-xml procedure, policies-dataset-read-xml procedure, and override-policies-from-xml procedure can be used together.

Calling API procedures to import audit policies in XML

```abl
{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 <> "":U 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. /* 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 the export-cached-audit-events procedure 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 the export-audit-events procedure 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).
Audit Policy Maintenance APIs

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 AppServer, if there is connection to the AppServer. 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 the
get-complete-dbname-list procedure, get-dbname-list procedure, 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 AppServer,
if there is a connection to an AppServer. 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 the `get-complete-dbname-list procedure`, `get-dbname-list procedure`, 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.
END.
```
isDBOnAppServer function

Checks if the database specified by a given database information string is on the AppServer. 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 the get-complete-dbname-list procedure, get-dbname-list procedure, 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.
END.
```
override-policies-from-xml procedure

This should only be called after a call to the policies-dataset-read-xml procedure 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 pErrorMsg AS CHARACTER**
  
  Error messages.

**Note:** For more information, see the “policies-dataset-read-xml procedure” section on page B–12.

**Example:** See the “cancel-import-from-xml procedure” section on page B–5.

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-OUTPUT DATASET-HANDLE FOR phDataSet**
  
  ProDataSet object.
- **OUTPUT pcList AS CHARACTER**
  
  List of existing policies.
- **OUTPUT pErrorMsg AS CHARACTER**
  
  Error messages.

**Example:** See the “cancel-import-from-xml procedure” section on page B–5.
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 AppServer specified by the caller and uses it whenever this procedure needs to forward a request to the AppServer.

Parameters:

INPUT phAppServer AS HANDLE

   Server handle for an AppServer 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 (1 of 2)

```abl
{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

```plaintext
{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 ttAuditEventPolicy: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 (dbname(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 <> "":U 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 <> "":U THEN
                RETURN ERROR errorMsg.
        END.
    END.
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** `pcErrMsg` 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 CHARACTER

Error messages.

getDataModified procedure

Checks if there are any pending changes in the local ProDataSet object (populated by the changeAuditDatabase procedure) 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
hasAppEvents 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 the saveAuditEventChanges procedure 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 the resubmit-import-from-xml procedure.

**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 the import-policies-from-xml procedure, 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 the `changeAuditDatabase procedure`.

**Parameters:** None

**Notes:**

- Check for any error messages using the ABL RETURN-VALUE function.
- This procedure calls the `update-auditing-policies procedure` 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 pcErrMsg 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 pcErrMsg 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 the import-xml-db procedure and save-changes-to-db procedure. For a ProDataSet, call the import-xml-fill-dset procedure and copy-changes-to-dset procedure. 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 the refresh-db-cache procedure 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** (1 of 2)

```ABL
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 */
        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.
```

```ABL
END.
END.
```
copy-changes-to-dset procedure

Fills the specified ProDataSet with the records loaded with the import-xml-fill-dset procedure. 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 policies 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 the save-changes-to-db procedure. 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.
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 the `copy-changes-to-dset procedure`. 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 the import-xml-db procedure or the save-changes-to-db procedure. It is up to the caller to decide if the cache should be refreshed after new policy settings are loaded. See the import-xml-db procedure and the save-changes-to-db procedure 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 the import-xml-db procedure 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 tMode 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 p1GetAllEvents 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 errorMsg 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.
HLC library functions provide an interface between your HLC functions and ABL (Advanced Business Language). This appendix contains the following sections:

- Function summary
- Function reference

For more information on using these functions, see Chapter 18, “Host Language Call Interface.”
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

Table C–1 lists HLC library functions that access ABL shared variables. Each function operates on variables with a specific ABL data type.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prordc()</td>
<td>Reads the value of a shared CHARACTER variable</td>
</tr>
<tr>
<td>prordd()</td>
<td>Reads the value of a shared DATE variable</td>
</tr>
<tr>
<td>prordi()</td>
<td>Reads the value of a shared INTEGER variable</td>
</tr>
<tr>
<td>prordl()</td>
<td>Reads the value of a shared LOGICAL variable</td>
</tr>
<tr>
<td>prordn()</td>
<td>Reads the value of a shared DECIMAL variable</td>
</tr>
<tr>
<td>prordr()</td>
<td>Reads the value of a shared RECID variable</td>
</tr>
<tr>
<td>prowtc()</td>
<td>Writes a value to a shared CHARACTER variable</td>
</tr>
<tr>
<td>prowtd()</td>
<td>Writes a value to a shared DATE variable</td>
</tr>
<tr>
<td>prowti()</td>
<td>Writes a value to a shared INTEGER variable</td>
</tr>
<tr>
<td>prowtd()</td>
<td>Writes a value to a shared LOGICAL variable</td>
</tr>
<tr>
<td>prowtn()</td>
<td>Writes a value to a shared DECIMAL variable</td>
</tr>
<tr>
<td>prowtr()</td>
<td>Writes a value to a shared RECID variable</td>
</tr>
</tbody>
</table>
Shared-buffer access

Table C–2 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 C–2: Functions that access ABL shared buffers

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>profldix()</td>
<td>Returns the handle to a specified field in a shared buffer</td>
</tr>
<tr>
<td>prordbc()</td>
<td>Reads the value of a CHARACTER field in a shared buffer</td>
</tr>
<tr>
<td>prordbd()</td>
<td>Reads the value of a DATE field in a shared buffer</td>
</tr>
<tr>
<td>prordbi()</td>
<td>Reads the value of an INTEGER field in a shared buffer</td>
</tr>
<tr>
<td>prordbl()</td>
<td>Reads the value of a LOGICAL field in a shared buffer</td>
</tr>
<tr>
<td>prordbn()</td>
<td>Reads the value of a DECIMAL field in a shared buffer</td>
</tr>
<tr>
<td>prordbr()</td>
<td>Reads the value of a RECID field in a shared buffer</td>
</tr>
<tr>
<td>prowtbc()</td>
<td>Writes a value to a CHARACTER field in a shared buffer</td>
</tr>
<tr>
<td>prowtbd()</td>
<td>Writes a value to a DATE field in a shared buffer</td>
</tr>
<tr>
<td>prowtbi()</td>
<td>Writes a value to an INTEGER field in a shared buffer</td>
</tr>
<tr>
<td>prowtbl()</td>
<td>Writes a value to a LOGICAL field in a shared buffer</td>
</tr>
<tr>
<td>prowtbn()</td>
<td>Writes a value to a DECIMAL field in a shared buffer</td>
</tr>
<tr>
<td>prowtbr()</td>
<td>Writes a value to a RECID field in a shared buffer</td>
</tr>
</tbody>
</table>
Screen display

Table C–3 lists HLC library functions that enable HLC applications to use the screen display in a manner consistent with ABL display standards.

Table C–3: Functions that interact with ABL displays

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>proscopn()</td>
<td>Restores a UNIX terminal to its I/O mode before OpenEdge was run</td>
</tr>
<tr>
<td>proclear()</td>
<td>Clears the display</td>
</tr>
<tr>
<td>prosccls()</td>
<td>Sets a UNIX terminal to raw mode</td>
</tr>
<tr>
<td>promsgd()</td>
<td>Displays a message on the display (in any I/O mode)</td>
</tr>
</tbody>
</table>

Interrupt handling

Table C–4 lists HLC library functions that allow applications to test for specific ABL interrupts.

Table C–4: Functions that test for ABL interrupts

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prockint()</td>
<td>Tests whether the interrupt key (STOP key) has been pressed</td>
</tr>
</tbody>
</table>

Timer-service routines

Table C–5 lists HLC library functions that access ABL-managed timer services.

Table C–5: Functions that access ABL timer services

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>prosleep()</td>
<td>Suspends execution for a specified time interval</td>
</tr>
<tr>
<td>proevt()</td>
<td>Sets a specified interval timer</td>
</tr>
<tr>
<td>provait()</td>
<td>Suspends execution until a specified interval timer (previously set by proevt()) expires</td>
</tr>
<tr>
<td>procncel()</td>
<td>Cancels a specified interval timer previously set by proevt()</td>
</tr>
</tbody>
</table>
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

```c
int prockint()
```

proclear() - Clear the Display

Clears the display.

On successful completion, proclear() returns 0; otherwise, it returns a non-zero value:

Syntax

```c
int proclear()
```

Notes:

- 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 Chapter 18, “Host Language Call Interface.”

See Also

promsgd() - Display Message, prosccls() - Set Terminal To Raw Mode and Refresh, proscopn() - Set Terminal To Initial Mode
**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**

```c
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 Chapter 18, “Host Language Call Interface.”

**See Also**

`prockint()` - Test for Interrupt Key, `proevt()` - Set Interval Timer, `prosleep()` - Sleep For Specified Interval, `prowait()` - Wait For Timer To Expire

**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**

```c
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 Chapter 18, “Host Language Call Interface.”

See Also

proncel() - Cancel Interval Timer, prockint() - Test for Interrupt Key, prosleep() - Sleep For Specified Interval, prowait() - Wait For Timer To Expire

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

```c
int profldix ( pbufnam, pfdnam )
    char *pbufnam;
    char *pfdnam;
```

*pbufnam*

Points to the name of the shared buffer containing the specified field. You supply the name from your OpenEdge application.

*pfdnam*

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, prordbd() - Read DATE Field, prordbi() - Read INTEGER Field, prordbl() - Read LOGICAL Field, prordbn() - Read DECIMAL Field, prordbr() - Read RECID Field, prowtbc() - Write CHARACTER Field, prowtbd() - Write DATE Field, prowtbi() - Write INTEGER Field, prowtbl() - Write LOGICAL Field, prowtbn() - Write DECIMAL Field, prowtbr() - Write RECID Field
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

```c
int
promsgd ( pmessage )
char *pmessage;
```

Notes:

- 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 Chapter 18, “Host Language Call Interface.”

- 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, `prosccls()` - Set Terminal To Raw Mode and Refresh, `proscopn()` - Set Terminal To Initial Mode
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

```c
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, prordbd() - Read DATE Field, prordbi() - Read INTEGER Field, prordbl() - Read LOGICAL Field, prordbn() - Read DECIMAL Field, prordbr() - Read RECID Field, prowtbc() - Write CHARACTER Field
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**

```c
int prordbd ( 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
- `prordbc()` - Read CHARACTER Field
- `prordbi()` - Read INTEGER Field
- `prordbl()` - Read LOGICAL Field
- `prordbn()` - Read DECIMAL Field
- `prordbr()` - Read RECID Field
- `prowtbd()` - Write DATE Field
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**

```c
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, `prordbc()` - Read CHARACTER Field, `prordbd()` - Read DATE Field, `prordbl()` - Read LOGICAL Field, `prordbn()` - Read DECIMAL Field, `prordbr()` - Read RECID Field, `prowtbi()` - Write INTEGER Field
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

```c
int prordbl ( 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.

- `punknown`:
  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
- prordbc() - Read CHARACTER Field
- prordbd() - Read DATE Field
- prordbi() - Read INTEGER Field
- prordbn() - Read DECIMAL Field
- prordbr() - Read RECID Field
- prowtbl() - Write LOGICAL Field
**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**

```c
int prordbn ( pbufnam, fhandle, index, pvar, punknown, varlen, pactlen, 
              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, `prordbc()` - Read CHARACTER Field, `prordbd()` - Read DATE Field, `prordb()` - Read INTEGER Field, `prordbl()` - Read LOGICAL Field, `prordbr()` - Read RECID Field, `prowtbvn()` - Write DECIMAL Field
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

```c
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, prordbc() - Read CHARACTER Field, prordbd() - Read DATE Field, prordbi() - Read INTEGER Field, prordbl() - Read LOGICAL Field, prordbn() - Read DECIMAL Field, prowtbr() - Write RECID Field
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**

```c
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, prordi() - Read INTEGER Variable, prordl() - Read LOGICAL Variable, prordn() - Read DECIMAL Variable, prordr() - Read RECID Variable, prowtc() - Write CHARACTER Variable
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

```c
int
prorrd ( 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 prorrd() returns the value of year.

**pmonth**

This output parameter points to an integer where prorrd() returns the value of month.

**pday**

This output parameter points to an integer where prorrd() returns the value of day.

**punknown**

This output parameter points to an integer where prorrd() returns 1 if the variable has the Unknown value (?), and returns 0 otherwise.

See Also

prordc() - Read CHARACTER Variable, prordi() - Read INTEGER Variable, prordl() - Read LOGICAL Variable, prordn() - Read DECIMAL Variable, prordr() - Read RECID Variable, prowtd() - Write DATE Variable
**prordi() - Read INTEGER Variable**

The `prordi()` function reads the value of a shared INTEGER variable.

On successful completion, `prordi()` returns 0; otherwise, it returns a non-zero value:

**Syntax**

```c
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`, `prordd() - Read DATE Variable`, `prordl() - Read LOGICAL Variable`, `prordn() - Read DECIMAL Variable`, `prordr() - Read RECID Variable`, `prowti() - Write INTEGER Variable`
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 prordl() 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, prordd() - Read DATE Variable, prordi() - Read INTEGER Variable, prordn() - Read DECIMAL Variable, prordr() - Read RECID Variable, prowtl() - Write LOGICAL Variable
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

```c
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, prordd() - Read DATE Variable, prordi() - Read INTEGER Variable, prordl() - Read LOGICAL Variable, prordr() - Read RECID Variable, prowtn() - Write DECIMAL Variable
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
- prordd() - Read DATE Variable
- prordi() - Read INTEGER Variable
- prordl() - Read LOGICAL Variable
- prordn() - Read DECIMAL Variable
- prowtr() - Write RECID Variable
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;
```

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 Chapter 18, “Host Language Call Interface.”

**See Also**
proclear() - Clear the Display, promsgd() - Display Message, proscopn() - Set Terminal To Initial Mode

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 Chapter 18, “Host Language Call Interface.”

**See Also**
proclear() - Clear the Display, promsgd() - Display Message, prosccls() - Set Terminal To Raw Mode and Refresh
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

```c
void prosleep ( seconds )
int seconds;
```

seconds

This input parameter specifies the number of seconds to suspend execution.

See Also

prockint() - Test for Interrupt Key, procncel() - Cancel Interval Timer, proevt() - Set Interval Timer, prowait() - Wait For Timer To Expire

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

```c
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 Chapter 18, “Host Language Call Interface.”

See Also

prockint() - Test for Interrupt Key, procncel() - Cancel Interval Timer, proevt() - Set Interval Timer, prosleep() - Sleep For Specified Interval
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

```c
int prowtbc ( 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, prordbc() - Read CHARACTER Field, prowtbd() - Write DATE Field, prowtbi() - Write INTEGER Field, prowtbl() - Write LOGICAL Field, prowtbn() - Write DECIMAL Field, prowtbr() - Write RECID Field
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

```c
int prowtbd ( pbufnam, fhandle, index, year, month, day,
              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, prordbd() - Read DATE Field, prowtdbc() - Write CHARACTER Field, prowtdbi() - Write INTEGER Field, prowtdbl() - Write LOGICAL Field, prowtdbn() - Write DECIMAL Field, prowtdbr() - Write RECID Field
Function reference

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**

```c
int prowtbi ( 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, `prordbi()` - Read INTEGER Field, `prowtbc()` - Write CHARACTER Field, `prowtbd()` - Write DATE Field, `prowtbl()` - Write LOGICAL Field, `prowtbn()` - Write DECIMAL Field, `prowtbr()` - Write RECID Field
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**

```c
int prowtbl ( pbufnam, fhandle, index, var, 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
- `proordbl()` - Read LOGICAL Field
- `prowtbc()` - Write CHARACTER Field
- `prowtbd()` - Write DATE Field
- `prowtbi()` - Write INTEGER Field
- `prowtbn()` - Write DECIMAL Field
- `prowtbr()` - Write RECID Field
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

```c
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, prordbn() - Read DECIMAL Field, prowtbc() - Write CHARACTER Field, prowtbd() - Write DATE Field, prowtbi() - Write INTEGER Field, prowtbl() - Write LOGICAL Field, prowtbr() - Write RECID Field
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**

```c
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
- `prordbr()` - Read RECID Field
- `prowtbc()` - Write CHARACTER Field
- `prowtd()` - Write DATE Field
- `prowti()` - Write INTEGER Field
- `prowtbl()` - Write LOGICAL Field
- `prowtn()` - Write DECIMAL Field
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

```c
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, provti() - Write INTEGER Variable, prowtl() - Write LOGICAL Variable, prowtn() - Write DECIMAL Variable, prowtr() - Write RECID Variable
**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**

```c
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
- `prowti()` - Write INTEGER Variable
- `prowtl()` - Write LOGICAL Variable
- `prowtn()` - Write DECIMAL Variable
- `prowtr()` - Write RECID Variable
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**

```c
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, prowtd() - Write DATE Variable, prowtl() - Write LOGICAL Variable, prowtn() - Write DECIMAL Variable, prowtr() - Write RECID Variable
prowtl() - Write LOGICAL Variable

The prowt1() function writes a Boolean value to a shared LOGICAL variable.

On successful completion, prowt1() returns 0; otherwise, it returns a non-zero value:

**Syntax**

```c
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, prowtd() - Write DATE Variable, prowti() - Write INTEGER Variable, prowtn() - Write DECIMAL Variable, prowtr() - Write RECID Variable
The prown() function writes a decimal value, formatted as a character string, to a shared DECIMAL numeric variable.

On successful completion, prown() returns 0; otherwise, it returns a non-zero value:

**Syntax**

```c
int prown ( 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, prowtd() - Write DATE Variable, prowti() - Write INTEGER Variable, prowtl() - Write LOGICAL Variable, prowtr() - Write RECID Variable
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**

```c
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, `prowtd()` - Write DATE Variable, `prowti()` - Write INTEGER Variable, `prowtl()` - Write LOGICAL Variable, `prowtn()` - Write DECIMAL Variable
Command and Utility Reference

This appendix contains the standard reference documentation for the 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:

<table>
<thead>
<tr>
<th>Operating system</th>
<th>Syntax</th>
</tr>
</thead>
</table>
| UNIX Windows     | quoter input-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 Table D–1 to run QUOTER from the Procedure Editor.

Table D–1: QUOTER examples

<table>
<thead>
<tr>
<th>Operating system</th>
<th>Using QUOTER—Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows</td>
<td>DOS quoter i-dat12.d &gt;i-dat12.q</td>
</tr>
<tr>
<td>UNIX</td>
<td>UNIX quoter i-datf12.d &gt;i-datf12.q</td>
</tr>
</tbody>
</table>

In Figure D–1, i-datf12.d is the name of the data file you supply to QUOTER while i-datf12.q is the name of the file in which you want to store QUOTER’s output.

![Diagram of how QUOTER prepares a file]

Figure D–1: How QUOTER prepares a file

The i-datf12.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-datf12.d) and without quotes (unlike i-datf1.d)? That is, your data file looks like i-datf13.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 NO-ATTR-SPACE 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-datfl3.q example.

   **i-datfl3.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 Figure D–2.

![Figure D–2: Extracting QUOTER input with SUBSTRING](image)

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-datfl4.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

```plaintext
OS-COMMAND SILENT quoter -d , i-datfl4.d >i-datfl4.q.
INPUT FROM i-datfl4.q NO-ECHO.
REPEAT:
  CREATE Customer.
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-datf14.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-datf15.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-datf15.d >i-datf15.q.
INPUT FROM i-datf15.q NO-ECHO.
REPEAT:
  CREATE Customer.
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.
Symbols

^ (caret), I/O statements 5–26
!
(exclamation point)
user ID lists 2–41
* (asterisk)
user ID lists 2–41
~ (tilde), input files 5–33

A

ABL data types
  converting to C data types 18–18
  converting to COM data types A–4
ABL data, accessing with HLC library
  functions 18–16
ABL key functions, ActiveX controls 16–16
Abnormal termination 18–25
Accessing help 10–9
Accessing OCXs at run time 16–12
ActiveX
  Automation 15–1
  Automation objects 6–7
  collections, navigating 14–15
  controls 6–7
ActiveX automation. See Automation
ActiveX controls (OCXs)
  accessing 8–21
  at run time 16–12
 assigning colors 6–7
 assigning fonts 6–7
 control container 16–2, 16–3
  See also Control-frame
 creating instances 16–7
 CSComboBox control 16–35
 CSSpin control 16–35
 defined 8–19, 16–1
 defining instances
  design-time properties 16–8
  invisible controls 16–11
  location 16–10
  naming 16–8, 16–9
  size 16–10
  tab order 16–11
 design mode, defined 8–22
 DLL files 8–23
 encapsulation 16–3
 event procedures in the AppBuilder
  16–32
 events 16–19
  See also Event procedures, OCX
  defined 8–21
 examples
  i-ocx1.w 16–2
 files
  design time 8–23
  run time 8–23
 handle connections 16–15
 installed with OpenEdge 16–35
 instance files (.wrx) 8–23
 instantiating at run time 16–12
 license information 8–23
 loading with ABL 16–12
 locating in viewer 14–20
 managing at run time 16–16
 ABL key functions 16–16
 releasing resources 16–17
 tab order and Z order 16–16
programming in the AppBuilder 16–25
See also AB and ActiveX controls
properties, design-time vs. run-time 16–7
PSTimer control 16–35
requirements 8–22
design mode 8–23
run mode 8–23
restrictions 16–6
run mode, defined 8–22
sources 8–22
ADD-EVENTS-PROCEDURE method
16–24
Adding records 1–17
Advise links
creating 13–13
defined 13–12
multiple conversations 13–15
removing 13–14
example 13–16
Aggregate functions
TOTAL BY 7–13
Aliases
compiling procedures 1–13
creating in applications 1–12
DICTDB 2–5
logical database name 1–11
shared record buffers 1–14
ALTERNATE-KEY option
SYSTEM-HELP statement 10–12
ANYWHERE option 4–27, 4–34
APIs, Audit Policy Management
access 3–21
AppBuilder 10–5, 10–7
AppBuilder and ActiveX controls
control loading and instantiation 16–28
control-frame instantiation 16–27
default handles 16–26
event procedures 16–31
initialization 16–31
resource definitions 16–25
standard procedures 16–34
applhelp.p 10–20
Application environment management 6–23
Application events
auditing 3–8
Application security 2–1
activities-based authorization 2–42
authorizing user IDs at run time 2–39
CAN-DO function 2–45, 2–46
creating, permissions table 2–42
cryptography 2–51
See also Cryptography
secure database connections 2–37
types 2–2
using the CAN-DO function 2–40
Application trusted domain registry 2–16
ABL for building 2–23
Application user ID
defined 2–7
managing 2–28
Archiving
audit 3–19
arrays A–5
AS Datatype option, COM object syntax
14–9
AS option 12–9
DEFINE PARAMETER statement 12–6
ASC function, code page conversion 5–27
ASSIGN statement 1–15
Asterisk (*)
user ID lists 2–41
Attributes
AVAILABLE-FORMATS 9–2
control-frame widget 16–4
DDE-ERROR 13–8
DDE-ID 13–8
DDE-ITEM 13–8
DDE-NAME 13–8
DDE-TOPIC 13–8
ITEMS-PER-ROW 9–3
MULTIPLE 9–3
NUM-FORMATS 9–3
security policy 2–52
TYPE 9–3
VALUE 9–3
Audit data tables 3–22
Audit Policy Management
API access 3–21
APIs B–1
AppServer usage B–4
caching B–16
exporting to XML B–24
generic utility B–5
importing from XML B–25
overview B–2
procedure file reference B–3
stand-alone procedures B–29
Audit policy tables 3–20
AUDIT-CONTROL system handle 3–9
AUDIT-ENABLED function 3–7
AUDIT-EVENT-CONTEXT attribute 2–12

Auditing
   developing tools 3–19
   audit archiving 3–22
   audit configuration 3–20
   recording application events 3–8
      ABL for 3–9
      planning 3–8
See also Auditing ABL

Auditing ABL 3–1
   asserting auditing ID 3–4
   audit archiving 3–22
      coding requirements 3–23
      required privileges and security 3–24
   audit configuration 3–20
      APIs for 3–21
      privileges required 3–22
   audit data table access 3–22
   audit event context
      application context 3–14
      application design 3–17
      audit event groups 3–15
      client login session 3–11
      management 3–11
   audit policy table access 3–20
   generating application events 3–9
   recording an application event 3–10
   security 3–2
      privileges 3–7
See also Auditing
   verifying audit enablement 3–7

Auditing ID
   asserting 3–4
      application design 3–6
      from database connection ID 3–4
      from OpenEdge session ID 3–5
   configuring 3–3
      defined 2–7
   establishing 3–2
   identity management 2–36

Auditing privileges 3–7

Authentication 2–2
   defined 2–3
   OpenEdge internal
      ABL-supported procedure 2–5
      examples 2–6
AUTHENTICATION-FAILED method 2–13

Authorization
   compile-time
      defined 2–2
   connection
      defined 2–2
   examples 2–39
   run-time 2–39
      activity access 2–42
      defined 2–2
   procedure execution 2–39
   table and field access 2–49
   schema
      defined 2–2

Auto-connect, connecting databases 1–4

Automation
   See also Automation objects
      defined 15–1
      examples, src\samples\ActiveX 15–10
      requirements 15–2
      servers, accessing 15–3

Automation objects
   accessing 8–21
      by name 15–4
      by name and file 15–6
      implied object and file 15–7
      named top-level object 15–5
   compared to DDE 8–18
      defined 15–1
   locating in viewer 14–19
   registry flags 15–3
   releasing 15–8
   requirements 8–22
   sources 8–22
   top-level, defined 15–2

Automation, ActiveX Automation objects 6–7

AVAILABLE-FORMATS attribute 9–2

B

Base fields 7–24

Base64 encoding 2–37

BASE64-DECODE function
   data size decrease 2–64
      defined 2–37
      managing crypto data with 2–63

BASE64-ENCODE function
   data size increase 2–64
      defined 2–37
      managing crypto data with 2–63

Batch processing 5–24

BEGIN-EVENT-GROUP method 3–9
   usage 3–15

BGCOLOR, widget assignment 6–5

Blank user IDs
   OpenEdge internal 2–5

Blocks
   editing 4–29
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>BROWSE statement</td>
<td>1–15</td>
</tr>
<tr>
<td>Buffers</td>
<td></td>
</tr>
<tr>
<td>record 1–15</td>
<td></td>
</tr>
<tr>
<td>screen 1–15</td>
<td></td>
</tr>
<tr>
<td>Building HLC executables. See HLC executable</td>
<td></td>
</tr>
<tr>
<td>Buttons, mouse</td>
<td>4–22</td>
</tr>
<tr>
<td>BY-POINTER type option</td>
<td>14–11</td>
</tr>
<tr>
<td>BYTE data type</td>
<td>12–10</td>
</tr>
<tr>
<td>running DLL routines</td>
<td>12–20</td>
</tr>
<tr>
<td>BY-VARIANT-POINTER type option</td>
<td>14–11</td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>C functions</td>
<td></td>
</tr>
<tr>
<td>declaration with FUNCTEST</td>
<td>18–11</td>
</tr>
<tr>
<td>HLC functions</td>
<td>18–16</td>
</tr>
<tr>
<td>naming</td>
<td>18–12</td>
</tr>
<tr>
<td>passing error codes</td>
<td>18–21</td>
</tr>
<tr>
<td>portability</td>
<td>18–12</td>
</tr>
<tr>
<td>returning error codes</td>
<td>18–11</td>
</tr>
<tr>
<td>writing</td>
<td>18–10</td>
</tr>
<tr>
<td>C language data types, converting to ABL data types</td>
<td>18–18</td>
</tr>
<tr>
<td>C source files, compiling</td>
<td>18–26</td>
</tr>
<tr>
<td>CALL statement</td>
<td></td>
</tr>
<tr>
<td>shared buffer</td>
<td>18–20</td>
</tr>
<tr>
<td>syntax</td>
<td>18–5</td>
</tr>
<tr>
<td>using</td>
<td>18–5</td>
</tr>
<tr>
<td>Calling conventions for DLL routines</td>
<td>12–6</td>
</tr>
<tr>
<td>Calling help</td>
<td>10–10</td>
</tr>
<tr>
<td>CAN-CREATE attribute</td>
<td>2–49</td>
</tr>
<tr>
<td>CAN-DELETE attribute</td>
<td>2–49</td>
</tr>
<tr>
<td>CAN-DO function</td>
<td></td>
</tr>
<tr>
<td>application security</td>
<td>2–45</td>
</tr>
<tr>
<td>using to authorize user IDs</td>
<td>2–41</td>
</tr>
<tr>
<td>using with ROLES attribute</td>
<td>2–42</td>
</tr>
<tr>
<td>CAN-READ attribute</td>
<td>2–49</td>
</tr>
<tr>
<td>CAN-WRITE attribute</td>
<td>2–49</td>
</tr>
<tr>
<td>CDECL option, PROCEDURE statement</td>
<td>12–6</td>
</tr>
<tr>
<td>CHARACTER data type</td>
<td>12–10</td>
</tr>
<tr>
<td>running DLL routines</td>
<td>12–20</td>
</tr>
<tr>
<td>when modified by shared library routine</td>
<td>12–17</td>
</tr>
<tr>
<td>Character interfaces</td>
<td></td>
</tr>
<tr>
<td>colors</td>
<td>6–9</td>
</tr>
<tr>
<td>reserved values</td>
<td>6–9</td>
</tr>
<tr>
<td>widget states</td>
<td>6–9</td>
</tr>
<tr>
<td>specifying colors</td>
<td>6–2</td>
</tr>
<tr>
<td>CHARACTER parameters, passing to shared library routines</td>
<td>12–17</td>
</tr>
<tr>
<td>Character set management. See Code page conversion</td>
<td></td>
</tr>
<tr>
<td>Checking data entry for keys</td>
<td>4–28</td>
</tr>
<tr>
<td>CHOOSE event</td>
<td>10–17, 10–18, 10–19, 10–20</td>
</tr>
<tr>
<td>CHR function</td>
<td></td>
</tr>
<tr>
<td>code page conversion</td>
<td>5–27</td>
</tr>
<tr>
<td>CLEAR-APPL-CONTEXT method</td>
<td>3–9</td>
</tr>
<tr>
<td>usage</td>
<td>3–14</td>
</tr>
<tr>
<td>Client login session</td>
<td></td>
</tr>
<tr>
<td>audit event context management</td>
<td>3–11</td>
</tr>
<tr>
<td>recording</td>
<td>2–17</td>
</tr>
<tr>
<td>requirements for using</td>
<td>2–16</td>
</tr>
<tr>
<td>starting and maintaining</td>
<td>2–17</td>
</tr>
<tr>
<td>Client principal object</td>
<td></td>
</tr>
<tr>
<td>handle</td>
<td>2–8</td>
</tr>
<tr>
<td>Client-principal object</td>
<td>2–9</td>
</tr>
<tr>
<td>attributes</td>
<td>2–10</td>
</tr>
<tr>
<td>optional</td>
<td>2–12</td>
</tr>
<tr>
<td>read-only</td>
<td>2–11</td>
</tr>
<tr>
<td>required</td>
<td>2–11</td>
</tr>
<tr>
<td>auditing usage</td>
<td>3–13</td>
</tr>
<tr>
<td>methods</td>
<td>2–13</td>
</tr>
<tr>
<td>login session management</td>
<td>2–13</td>
</tr>
<tr>
<td>set and get</td>
<td>2–13</td>
</tr>
<tr>
<td>n-tier applications</td>
<td>2–30</td>
</tr>
<tr>
<td>CLIENT-TTY attribute</td>
<td>2–12</td>
</tr>
<tr>
<td>CLIENT-WORKSTATION attribute</td>
<td>2–12</td>
</tr>
<tr>
<td>Clipboard. See System clipboard</td>
<td></td>
</tr>
<tr>
<td>Clipboard examples</td>
<td></td>
</tr>
<tr>
<td>i-clpbrd.p 9–8</td>
<td></td>
</tr>
<tr>
<td>i-clpmul.p 9–14</td>
<td></td>
</tr>
<tr>
<td>Clipboard output</td>
<td>5–6</td>
</tr>
<tr>
<td>CLIPBOARD system handle</td>
<td>9–2</td>
</tr>
<tr>
<td>example procedure</td>
<td></td>
</tr>
<tr>
<td>multiple-item transfer</td>
<td>9–14</td>
</tr>
<tr>
<td>single-item transfer</td>
<td>9–8</td>
</tr>
<tr>
<td>multiple-item transfers</td>
<td>9–12</td>
</tr>
<tr>
<td>requirements</td>
<td>8–12</td>
</tr>
<tr>
<td>single-item transfers</td>
<td>9–4</td>
</tr>
</tbody>
</table>
Code pages
conversion map file 5–27
conversions 5–27
example of conversion 5–28
CODEPAGE-CONVERT function 5–27
Coding help calls
10–15
See also Help calling interface
Coding help strings 10–4
Collections, navigating 14–15
Color
adding for startup 6–3
availability 6–2
character interfaces 6–9
See also Character interfaces
management 6–9
reserved values 6–9
widget states 6–9
common dialog 6–13
example 6–13
custom, Windows color dialog 6–14
definitions 6–21
changing 6–21
dynamic 6–13
environment files 6–2
See also Environment files
evironments 6–23
graphical interfaces
definition access 6–21
environment management 6–23
management 6–12
table access 6–19
user changes 6–13
inheritance 6–8
managing display limitations 6–26
numeric, Windows color dialog 6–14
OpenEdge defaults 6–3
RGB-VALUE function 6–2
table access 6–19
the registry 6–2
visual, Windows color dialog 6–14
widget assignment 6–5
example 6–5
Windows 6–11
Palette Manager 6–11
Color dialogs 6–13
ABL statements 6–13, 6–15
Windows example 6–13
Colors
assigning 6–5
assigning to ActiveX controls 6–7
modifying 6–19
COLOR-TABLE handle 6–19
example 6–6, 6–13
using 6–19
COM data types
alternate names A–7
converting to ABL A–3
COM Object Viewer 14–18
accessing Type Libraries 14–18
locating
ActiveX controls 14–20
Automation objects 14–19
methods, properties, and events 14–21
COM objects
See also ActiveX controls (OCXs),
Automation objects
accessing 8–21
ActiveX collections 14–15
colors and fonts 14–14
compared to widgets 14–2
features 14–3
memory management 14–16
control-frame 16–3
defined 8–18
error handling 14–17
handles 14–2
managing 14–14
memory resources 14–16
obtaining access 14–4
properties and methods 14–5
See also Properties and methods,
COM object
sources 8–22
Type Libraries 14–10
viewing on-line 14–18
See also COM Object Viewer
COM-HANDLE data type 14–2
COMMAND option
SYSTEM-HELP statement 10–14
Common dialogs
ABL statements 6–13, 6–15, 6–17
color 6–13
eamples 6–13, 6–16
font 6–16
Compile-time, authorization 2–2
Component handles 14–2
accessing for OCXs 16–13
ActiveX control connections 16–15
expressions 14–6
releasing COM objects 14–16
validating 14–14
COM-SELF system handle 16–23
Configuration
audit 3–19
CONNECT statement 1–3
authentication 2–6
NO-ERROR option 1–3, 1–6
Index–5
CONNECTED function 1–6

Connection authorization  
  defined 2–2

Connection failures 1–5  
  See also Multi-database connection(s)

Connection overhead 1–8

Connection parameters. See Startup parameters

CONTAINS operator 1–41

CONTENTS option  
  SYSTEM-HELP statement 10–12

CONTEXT option  
  SYSTEM-HELP statement 10–12

CONTEXT-POPUP option  
  SYSTEM-HELP statement 10–13

Context-sensitive help 10–9, 10–19, 10–20

Control container, ActiveX, defined 16–2

Control sequences 7–34

Control tables, sequences 1–54

Control-frame 16–3  
  attributes and properties 16–4  
  COM object 16–3  
  widget 16–3  
  events 16–19

Controllers, Automation, defined 15–1

Control-Name property, OCX handle access 16–14

Controls  
  ActiveX controls 6–7

Conversations, DDE  
  closing 13–16  
  defined 13–3  
  end-point 13–7  
  exchanging data 13–12  
  opening 13–9  
  structure 13–5  
  using 13–3

Conversion map files 5–27

Converting  
  code pages 5–27  
  See also Code page conversion  
  data 5–29  
  IMPORT statement 5–32

Cooked terminal I/O mode 18–23

Copy operations using the clipboard 9–7

CREATE ALIAS statement 1–11

CREATE Automation Object statement 15–3

CREATE CLIENT-PRINCIPAL statement 2–8

CREATE statement 1–15

Crypto data 2–61  
  database and transport management 2–63  
  management requirements 2–62  
  size increase 2–63  
  storage 2–62

Cryptography 2–51  
  algorithms  
    symmetric encryption 2–53  
  attributes 2–52  
  code examples 2–58  
  defined 2–2  
  encryption keys  
    generating 2–54  
    setting symmetric 2–54  
  generating keys 2–54  
  key generation functions 2–55  
  managing crypto data 2–61  
  message digests 2–60  
  security policy 2–52  
  symmetric functions

CSComboBox control (OCX) 16–35

CSSpin control (OCX) 16–35

CURRENCY type specifier 14–9

CURRENT-VALUE function 1–60  
  syntax 1–58

CURRENT-VALUE statement 1–61  
  syntax 1–58

Cursor repositioning  
  field lists 1–32  
  FIND statement 1–27  
  after FIND fetches 1–27

Custom colors, Windows color dialog 6–14

Cut operations using clipboard 9–6

D

Data  
  cryptography 2–51  
  See also Cryptography  
  handling statements 1–15  
  movement 1–15
Data Dictionary 10–4
creating, permissions table 2–42
Data entry, monitoring 4–2, 4–28
Data integrity
Data size 18–15
Data type compatibilities
DLL parameters with C and MS-Windows 12–10
RUN parameters with DLL parameters 12–20
Data type conversions
COM object A–1
ABL to COM data types A–4
COM to ABL data types A–3
features and limitations A–2
pointers A–5
strategy A–2
types not converted A–6
HLC 18–18
Data types
COM alternate names A–7
COM object syntax 14–9
COM-HANDLE 14–2
DLL RUN parameter lists 12–20
parameters, OCX event procedures 16–21
RAW 1–64
shared library parameter definitions 12–9
Database connection ID
defined 2–7
managing 2–19
from external authentication system 2–21
from internal authentication system 2–19
OpenEdge internal rules for forming 2–4
Database events
auditing 3–8
Database fields
help 10–3
help strings 10–2
Database trusted domain registry 2–16
Databases
connect parameters, limits 1–2
connection security 2–37
multiple connections. See Multi-database connections
Number of Databases (-h) startup parameter 1–2
security 2–1
DCOLOR
using 6–9
widget assignment 6–5
widget states 6–9
DDE. See Dynamic data exchange
DDE ADVISE statement
purpose 13–4
using 13–13
DDE EXECUTE statement
purpose 13–4
using 13–12
DDE frames
DDE-NOTIFY events 13–13
defined 13–7, 13–10
DDE GET statement
purpose 13–4
using 13–13
DDE INITIATE statement
purpose 13–4
using 13–11
DDE REQUEST statement
purpose 13–4
using 13–12
DDE Return Codes 13–9
DDE SEND statement
purpose 13–4
using 13–12
DDE server preparations 13–9
DDE TERMINATE statement
purpose 13–4
using 13–16
DDE-ERROR attribute 13–8
DDE-ID attribute 13–8
DDE-ITEM attribute 13–8
DDE-NAME attribute 13–8
DDE-NOTIFY events
DDE frames 13–7
trapping 13–13
DDE-TOPIC attribute 13–8
Decrypt function
code examples 2–58
parameters 2–57
syntax 2–56
Index

Decryption
   See Cryptography

Default properties and methods 14–13
DEFINE BROWSE statement 10–7
DEFINE BUTTON statement 10–7
DEFINE FRAME statement 10–4
DEFINE IMAGE statement 10–7
DEFINE PARAMETER statement for DLLs 12–5
DEFINE QUERY statement 1–25
   SCROLLING option 1–25
DEFINE RECTANGLE statement 10–7
DEFINE SHARED STREAM statement 5–20
   See also Streams, shared
DEFINE STREAM statement 5–17
Defining shared library parameters 12–5
DELETE ALIAS statement 1–12
DELETE statement 1–15, 1–17
DELETE WIDGET statement, releasing ActiveX controls 16–17
Deleting
   field lists 1–31
   records 1–17
DELETER option
   EXPORT statement 5–30
   IMPORT statement 5–31, 5–32, 5–34
Demand-driven DDE 13–12
Design mode, ActiveX control 8–22
   requirements 8–23
Design-time OCX properties 16–7
   setting 16–8
Dialog boxes
   help 10–2
DICTDB alias 2–5
Directories, HLC 18–9
DISCONNECT statement
   affect on aliases 1–11
Dispatch routine, HLC. See PRODSP
   dispatch routine 18–6
DISPLAY statement 1–15
Displaying data 7–9
DLL. See Dynamic link library (DLL)
DLL examples
   i-dllex1.p 12–23
   i-dllex2.p 12–23
DLL routines
   passing NULLs 12–15
DLL structures
   allocating with DLL routines 12–16
   freeing memory 8–4, 12–17
   functions for reading 8–7
   statements for writing 8–6
DOMAIN-DESCRIPTION attribute 2–12
DOMAIN-NAME attribute 2–12
DOMAIN-TYPE attribute 2–12
DOUBLE data type 12–10
   running DLL routines 12–20
Dynamic Data Exchange (DDE) 8–16
   ABL statements 13–4
   application name 13–5
   channel number
      defined 13–5
      usage 13–11
   client, defined 13–3
   client/server coordination 13–14
   conversational exchange
      See also Exchange, conversational
      defined 13–7
      types 13–12
   conversations
      closing 13–16
      defined 13–3
      exchanging data 13–12
      hierarchy 13–5
      multiple 13–15
      opening 13–9
      structure 13–5
   data item name 13–5
   DDE-NOTIFY events 13–13
      defined 8–16, 13–1
      demand-driven 13–12
      event-driven 13–13
      example procedure 13–17
      frame attributes 13–7
      frames
         DDE-NOTIFY events 13–13
            defined 13–3, 13–7, 13–10
            initiating 13–11
            overview 13–3
            requirements 8–17
      server
         defined 13–3
         preparation 13–9
topic name 13–5
using 13–3

Dynamic link libraries (DLL) 8–14, 18–3
access to ABL widgets 12–18
capabilities 12–1
defined 8–14, 12–1
HWND attribute 12–18
NULL parameter values 12–15
parameter definition 12–5
requirements 8–15
using 12–2

defined 16–19
examples 16–23
AppBuilder 16–31
parameters 16–21
data type conversions A–1
searching at run time 16–24
specifying in the UIB 16–20
system handles 16–23

Event-driven DDE 13–13
applications 13–14

Events
ActiveX control 16–19
defined 8–21
locating in viewer 14–21
control-frame widget 16–19
keyboard 4–28
widget and OCX
applying from ABL 16–23

Example applications
ActiveX control 16–35
Automation 15–10

Example procedures
Clipboard multiple-transfer 9–14
Clipboard single-transfer 9–8
dll for message display 12–20
dll to play sounds 12–20
UNIX named pipe 11–7

Exchange, conversational
defined 13–7
demand-driven
defined 13–12
example 13–13
using 13–12
event-driven
defined 13–12
example 13–14
using 13–13
types 13–12

Exclamation point (!)
user ID lists 2–41

Exiting help from application 10–21

EXPORT statement 5–30
DELIMITER option 5–30

Exporting data 5–30

EXPORT-PRINCIPAL method 2–14

EXTRACT mouse button 4–22

External authentication systems 2–3
managing database connection ID 2–21

External program interfaces (EPIs), defined
8–1

Environment files
changing resources 6–21
colors and fonts 6–2
hints for managing 6–25
managing 6–23
managing multiple 6–24
managing statements 6–23

EPI. See External program interfaces (EPIs)
8–1

Error codes, returning 18–11

Error handling
COM objects 14–17
UNIX Systems 18–25

ERROR-CODE type specifier 14–9

Errors, DDE 13–8

Event procedures, OCX
coding rules 16–20

Encoding UUIDs 2–37

ENCAPT function
code examples 2–58
parameters 2–57
syntax 2–56

Encryption
See Cryptography

ENCAPTION-SALT attribute 2–52

END-ERROR key 4–26

END-EVENT-GROUP method 3–9
usage 3–15

Environment files
changing resources 6–21
colors and fonts 6–2
hints for managing 6–25
managing 6–23
managing multiple 6–24
managing statements 6–23

EPI. See External program interfaces (EPIs)
8–1

Error codes, returning 18–11

Error handling
COM objects 14–17
UNIX Systems 18–25

ERROR-CODE type specifier 14–9

Errors, DDE 13–8

Event procedures, OCX
coding rules 16–20

defined 16–19
examples 16–23
AppBuilder 16–31
parameters 16–21
data type conversions A–1
searching at run time 16–24
specifying in the UIB 16–20
system handles 16–23

Event-driven DDE 13–13
applications 13–14

Events
ActiveX control 16–19
defined 8–21
locating in viewer 14–21
control-frame widget 16–19
keyboard 4–28
widget and OCX
applying from ABL 16–23

Example applications
ActiveX control 16–35
Automation 15–10

Example procedures
Clipboard multiple-transfer 9–14
Clipboard single-transfer 9–8
dll for message display 12–20
dll to play sounds 12–20
UNIX named pipe 11–7

Exchange, conversational
defined 13–7
demand-driven
defined 13–12
example 13–13
using 13–12
event-driven
defined 13–12
example 13–14
using 13–13
types 13–12

Exclamation point (!)
user ID lists 2–41

Exiting help from application 10–21

EXPORT statement 5–30
DELIMITER option 5–30

Exporting data 5–30

EXPORT-PRINCIPAL method 2–14

EXTRACT mouse button 4–22

External authentication systems 2–3
managing database connection ID 2–21

External program interfaces (EPIs), defined
8–1

Index

F

Fetching records 1–19
   defining record set 1–18
   order 1–19
   preselect list 1–20
   results list 1–20
   ROWID and RECID 1–20
   using field lists. See Field lists

FGCOLOR, widget assignment 6–5

Field List Disable (-fldisable) startup parameter 1–33

Field lists 1–29
   benefits 1–29
   cursor repositioning 1–32
   DataServers 1–29, 1–32, 1–33
   degenerate cases 1–33
   deletes 1–31
   deployment 1–33
   implied entries 1–31
   multiple queries 1–33
   server versions 1–33
   specifying 1–30
   updates 1–31

Fields
   validation 5–29

File types 5–25

FILE-INFO system handle 5–26

Files
   ActiveX control 8–23
   attributes 5–25
   conversion map 5–27
   convmap.cp 5–27
   HLC 18–9
   information 5–26
   input 5–29
   redirecting input 5–13
   redirecting output 5–5
   schema cache 1–69
   See also Schema cache files

FIND statement 1–15
   repositioning 1–27
   after FIND fetches 1–27

FINDER option
   SYSTEM-HELP statement 10–14

FLOAT data type 12–10
   running DLL routines 12–20

FLOAT type specifier 14–9

FOCUS handle, clipboard 9–4

Font dialogs 6–16
   ABL statements 6–13, 6–17
   Windows example 6–16

FONT, widget assignment 6–5

Fonts
   adding for startup 6–4
   assigning 6–5
   assigning to ActiveX controls 6–7
   availability 6–2
   character interfaces, management 6–9
   common dialog 6–13
      example 6–16
      definitions 6–21
      changing 6–21
   environment files 6–2
   See also Environment files
   environments 6–23
   graphical interfaces
      definition access 6–21
      environment management 6–23
      management 6–12
      table access 6–19
      user changes 6–13
   inheritance 6–8
   OpenEdge defaults on Windows 6–4
   table access 6–19
   widget assignment 6–5
      example 6–5

FONT-TABLE handle 6–19
   example 6–6
   using 6–20

FOR EACH statement 1–15

FORCE-FILE option
   SYSTEM-HELP statement 10–14

FORM statement 10–4

Format phrase 10–4

ABL elements, for sockets 17–2

Frame backgrounds 7–25

Frame headers 7–25

FRAME-FIELD function 4–34

FUNCTEST macro
   C function declaration 18–11
   syntax 18–7

Function keys
   changing definitions 4–20
   monitoring usage 4–28

Functions
   ASC 5–27
   CAN-DO 2–39, 2–43, 2–45
   CHR 5–27
CODEPAGE-CONVERT 5–27
ENCODE 2–4
ENCRYPT and DECRYPT
FRAME-FIELD 4–34
GENERATE-PBE-KEY 2–55
GENERATE-PBE-SALT 2–55
GENERATE-RANDOM-KEY 2–56
HLC interrupt handling C–4
HLC library C–5
HLC screen display C–4
HLC shared buffer access C–3
HLC shared variable access C–2
HLC timer service routines C–4
KBLABEL 4–3
KEYCODE 4–3, 4–4
KEYFUNCTION 4–3
KEYLABEL 4–3, 4–4
LASTKEY 4–25
OS-GETENV 5–26
PAGE-NUMBER 7–26
SEARCH 5–7
SETUSERID 2–5
SUBSTRING D–5
USERID 2–41

G

Garbage collection, COM objects 14–16
GENERATE-PBE-KEY function 2–55
GENERATE-PBE-SALT function 2–55
GENERATE-RANDOM-KEY function 2–56
GENERATE-UUID function 2–36
GET-BYTE function 1–65
GET-datatype functions 8–5
GET-KEY-VALUE statement 6–21
example 6–21
GET-POINTER-VALUE function 8–10
GET-PROPERTY method 2–13
GET-RGB-VALUE method, COM objects 14–15
GET-SIZE function 8–4
Global unique identifier creating 2–36
GO-ON phrase 4–24
GUID function 2–36

H

HANDLE option 12–10, 12–11
Handles
COLOR-TABLE 6–19
FILE-INFO 5–26
FONT-TABLE 6–20
SECURITY-POLICY 2–9
application trusted domain registry 2–23
cryptography 2–52

handles 5–21
Help access methods 10–17, 10–20
Help buttons 10–20
Help calling interface 10–10
coding 10–15
HELP events 10–20, 10–21
Help keys 10–20
HELP option 10–4
SYSTEM-HELP statement 10–15
Help strings
coding in the AppBuilder 10–5
database fields 10–3
dialog boxes 10–2
field-level widgets 10–4
in code 10–4
Help systems 10–9
context-sensitive help 10–9
Help topics 10–10
HELP-TOPIC option
SYSTEM-HELP statement 10–12
HEX-DECODE function
defined 2–37
managing crypto data with 2–63
HEX-ENCODER function
defined 2–37
managing crypto data with 2–63
Hexidecimal encoding 2–37
HLC 18–3
common errors 18–13
directories 18–9
filenames 18–9
mapping C functions 18–6
UNIX Systems 18–23
using 18–3
HLC datatype conversions 18–18

HLC dispatch routine. See PRODSP dispatch routine 18–6

HLC executables, building 18–4, 18–22

HLC library functions C–5
  calling 18–19
  example 18–18
  guidelines for accessing ABL data 18–16
  interrupt handling C–4
  screen display C–4
  shared buffer 18–20
  shared buffer access C–3
  shared variable access C–2
  timer service routines C–4

HLC library functions described
  prockint() C–5
  proclear() C–5
  proncel() C–6
  proevt() C–6
  profdix() C–7
  promsgd() C–8
  prordbc() C–9
  prordbd() C–10
  prordbi() C–11
  prordbl() C–12
  prordbn() C–13
  prordbr() C–14
  prordc() C–15
  prordd() C–16
  prordi() C–17
  prordl() C–18
  prordn() C–19
  prordr() C–20
  prosccls() C–21
  proscoopn() C–21
  prosleepl() C–22
  proswait() C–22
  promthtc() C–23
  provtbd() C–24
  provtbi() C–25
  provtbl() C–26
  provtbn() C–27
  provtbr() C–28
  provtcc() C–29
  provtd() C–30
  provti() C–31
  provtl() C–32
  provtn() C–33
  provtr() C–34

Host Language Call interface 18–2, 18–3
  requirements 18–2

HWND attribute 12–18

I/O with other processes 5–23

Identity management 2–2, 2–7
  ABL to manage user IDs 2–8
  application trusted domain registry 2–23
  application user IDs
    managing 2–28
    auditing IDs
      managing 2–36
    client login sessions 2–16
    client-principal objects 2–9
    database connection IDs 2–19
    n-tier applications 2–28
      AppServer operating mode 2–29
      client-principal object 2–30
      state-free 2–33
      stateless 2–31
    OpenEdge session IDs 2–23
    security policy 2–23
    trusted domain registries 2–16
    unique object IDs 2–36

IMPORT statement 5–32
  DELIMITER option 5–31, 5–32, 5–34

Importing, data 5–32

IMPORT-PRINCIPAL method 2–15

Index cursors 1–27

Indexing, word 1–41

Index-sorting queries 1–23

Initialization files, colors and fonts 6–2

Inner joins
  defined 1–35
  example 1–34, 1–37
  mixing with left outer 1–40
  specifying 1–35

Input
  changing the source 5–13
  control character 5–16
  devices 5–14
  directory contents 5–14
  file preparation 5–29
    IMPORT statement 5–32
  files 5–13
  multiple sources 5–14, 5–17
  streams 5–2
  terminal 5–14
  working within procedures 5–1

INPUT CLOSE statement 5–16
  QUOTER utility D–5

Input focus 10–20
INPUT FROM statement 5–13
  BINARY option 5–16
code page conversion 5–27
  OS-DIR option 5–25
  QUOTER utility D–4
redirecting input 5–15

INPUT mode option, OCX event parameters 16–23

INPUT THROUGH statement 5–23
code page conversion 5–27

INPUT-OUTPUT mode option
method parameters 14–12
OCX event parameters 16–23

INPUT-OUTPUT THROUGH statement 5–23
code page conversion 5–27

INSERT statement 1–15, 1–17

Instantiating ActiveX controls 16–12

Internal authentication system 2–3
  managing database connection ID 2–19

Internal events
  auditing 3–8
  internationalization, word break tables 1–41

Interprocess communications
  I/O with other processes 5–23

Interrupt handling, HLC library functions C–4

Invisible ActiveX controls 16–11

Item property, OCX handle access 16–14

ITEMS-PER-ROW attribute 9–3
  example 9–14

IUNKNOWN type specifier 14–9

K

KBLABEL function 4–3

Key functions. See ABL key functions

KEY option
  SYSTEM-HELP statement 10–12
  -keyalias connection parameter
    socket servers 17–20
  -keyaliaspsswd connection parameter
    socket servers 17–20
  Keyboard events 4–28

KEYCODE function 4–3, 4–4

KEYFUNCTION function 4–3

KEYLABEL function 4–3, 4–4

Keys
  changing the function 4–20
codes 4–5, 4–6, 4–15
  END-ERROR 4–26
  functions 4–16, 4–28
  input 4–2
  input precedence 4–3
  labels 4–5, 4–6, 4–15, 4–28

Keystrokes
  monitoring 4–2, 4–28

L

LASTKEY function 4–25

Left outer joins
  defined 1–35
  example 1–36, 1–38
  mixing with inner 1–40
  specifying 1–35

Licensed user count, exceeding 1–2

LIKE option 12–11
  connected database 1–68
  DEFINE PARAMETER statement 12–6

LIST-PROPERTY-NAMES method 2–13

LOAD statement
  using 6–23

LoadControls method, role in OCX instantiation 16–12

LOAD-DOMAINS method 2–24

LOCK-REGISTRATION method 2–24
LOG-AUDIT-EVENT method 3–9
  usage 3–10
_login.p procedure 2–4, 2–5

LOGIN-EXPIRATION-TIMESTAMP
  attribute 2–12
LOGIN-HOST attribute 2–13
LOGIN-STATE attribute 2–11
LOGOUT method 2–15
LONG data type 12–10
  running DLL routines 12–20

M

Managing audit event context 3–11
Managing COM objects 14–14
Memory allocation, C routines 18–14
Memory management, COM objects 14–16
MEMPTR data type 12–9, 12–10
  database fields 12–11
  referencing strings 12–17
  running DLL routines 12–20
  running shared library routines 12–16
MEMPTR parameters, used to pass
  character strings 12–17
MEMPTR variables
  initialization 8–4
  initializing and uninitializing 8–3
  DLL routines 12–16
  obtaining the pointer value 8–10
  passing string values 12–17
  reading and writing 8–5
  using 12–16
MENU mouse button 4–22

Menus
  accessing help 10–17
Message digests 2–60
  See also Cryptography

Method references, COM object 14–5
  parentheses required 14–13
  restrictions 14–13
Methods and properties, COM object. See
  Properties and methods, COM object

Methods, COM object
  See also Properties and methods, COM

object
  passing parameters 14–10
  named 14–13

Mode options
  purpose 14–10
  using 14–12

Monitoring keystrokes 4–2, 4–28
  EDITING blocks 4–28
  user interface triggers 4–28

Mouse
  buttons 4–22
    EXTEND 4–22
    MENU 4–22
    MOVE 4–22
  events
    specifying 4–23
    input 4–2

Mouse buttons, SELECT 4–22

MOVE mouse button 4–22
Multi-database applications
  referencing files and field names 1–66

Multi-database connections
  ABL functions 1–7
  CONNECTED function 1–6
  connection overhead 1–8
  connection parameters 1–2
  conserving 1–8
  considerations at development 1–5
  disconnecting 1–9
  failures and disruptions 1–5
  logical database name 1–10
  methods list 1–2
  Physical Database Name (-db) startup
    parameter 1–3
  positioning references in applications
    1–67
  run-time considerations 1–5

Multi-databases
  disconnecting 1–9
  logical database name 1–9

Multiple
  input sources 5–14
  output destinations 5–7
MULTIPLE attribute 9–3
  example 9–14

Multiple-item transfers, clipboard 9–12
  data-based 9–13
  example 9–14
  widget-based 9–12
MULTIPLE-KEY option
Index

MULTITASKING-INTERVAL attribute using 13–14

N
NAME attribute, control-frame 16–9
Name property, ActiveX controls 16–8
Named pipes
accessing 11–2, 11–6
advantages 11–4
applications 11–2
capacity 11–3
compared to files and unnamed pipes 11–2
creating
mknod command 11–6
mknod() system call 11–6
defined 8–13, 11–1
deleting
rm command 11–6
unlink() system call 11–6
disadvantages 11–4
examples
i-do-sql.p 11–7
i-pipex2.p 11–7
I/O synchronization 11–3
interleaving messages 11–3
message handler procedure 11–2
multiple users 11–3
operational characteristics 11–3
overview 11–2
process blocking 11–3
request format example 11–11
requirements 8–13
scenario 11–2
UNIX
creating 11–5
deleting 11–6
examples 11–7
using 11–5
writing messages 11–3
Naming ActiveX controls 16–8
Naming control-frames 16–9
NEXT-VALUE function 1–60
syntax 1–58
NO-ERROR option
CONNECT statement 1–3
-nohostverify connection parameter
database clients 2–38
socket clients 17–21
Numer of Databases (-h) startup parameter 1–2
Number of Users (-n) startup parameter 1–5
Numeric colors, Windows color dialog 6–14
NUM-FORMATS attribute 9–3

O
Objects
Automation 15–1
COM 8–18
OCX. See ActiveX controls (OCXs)
OCX event triggers. See Event procedures, OCX
OCX examples
i-ocx1.w 16–25
ON statement
ANYWHERE option 4–27, 4–34
changing key function 4–20
OpenEdge
internal authentication system 2–3
OpenEdge database functions 1–7
OpenEdge file, colors and fonts 1–7
OpenEdge session ID
defined 2–7
managing 2–23
setting 2–25
ORDINAL option, PROCEDURE
statement 12–6
OS-DIR option, INPUT FROM statement 5–25
OS-GETENV function 5–26
_osprint.p 5–10
Outer joins
see also Left outer join
full outer 1–39
right outer 1–39
Output
changing the destination 5–3
devices 5–5
files 5–5
multiple destinations 5–7, 5–17
printers 5–3
PUT statement 5–31
  streams 5–2
  terminal 5–6
  Windows clipboard 5–6
  working within procedures 5–1

OUTPUT CLOSE statement 5–8, 5–19

Output destinations 7–20

OUTPUT mode option
  method parameters 14–12
  OCX event parameters 16–23

OUTPUT STREAM statement 5–18

OUTPUT THROUGH statement 5–23
  code page conversion 5–27

OUTPUT TO statement 5–3, 5–8, 7–20,
  7–32
  code page conversion 5–27
  PAGED option 5–5

Output, redirecting 7–20

P

PAGE-BOTTOM option 7–26

PAGED option, OUTPUT TO statement 5–5

PAGE-NUMBER function 7–26

Parameter files
  different systems 1–2
  invoking 1–2
  Parameter File (-pf) startup parameter 1–2

Parameters
  definitions for shared library routines 12–5
  OCX event procedure 16–21
  data type conversions A–1
  mode option considerations 16–22
  passing
    COM object methods 14–10
    named 14–13
    passing to DLL RUN statements 12–19

PARTIAL-KEY option
  SYSTEM-HELP statement 10–14

PASCAL option, PROCEDURE statement 12–6

Password
  ENCODE function 2–4
  OpenEdge internal rules for forming 2–4
  validating 2–4
  Paste operations using clipboard 9–6
  PBE-HASH-ALGORITHM attribute 2–52
  PBE-KEY-ROUNDS attribute 2–52

Permissions table 2–42
  adding records 2–42
  security 2–47
  using the Data Dictionary to create 2–42

PERSISTENT option, PROCEDURE statement 12–5

PFCOLOR
  using 6–9
  widget assignment 6–5
  widget states 6–9

Physical Database Name (-db) startup parameter, connection parameter 1–2

Pointers, method parameters 14–11

POSITION MAXIMIZE option
  SYSTEM-HELP statement 10–13

POSITION X Y WIDTH HEIGHT option
  SYSTEM-HELP statement 10–13

Positioning ActiveX controls 16–10

Preparing input files 5–29

Preselect lists. See Results lists

Preselected queries 1–23

Presorted queries 1–23

Printers
  redirecting output 5–3
  setting up in Windows 5–4, 5–5

Printing
  _osprint.p 5–9
  converting widgets 5–9
  solutions 5–9
  text files 5–9

PROCEDURE statement for DLLs
  C, Pascal, and standard conventions 12–6
  options 12–5
  routines declared by number 12–6

PROCEDURE statement for shared libraries 12–5
  parameter data types 12–9

Procedures
  changing output destination 5–3
  changing the input source 5–13
  continuing processing 4–24
monitoring keystrokes 4–2
sharing streams 5–20
using for activity authorization 2–43

Processes, input and output streams 5–23

PRODSP dispatch routine
defined 18–6
hlpodsp.c source file 18–6
mandatory declaration 18–6
mapping C functions 18–6
mapping routine identifiers 18–6
modifying the prototype file 18–6

Programming, multi-database, techniques, and considerations 1–66

PROMPT-FOR statement 10–4, 1–15

Properties
ActiveX control access 16–13
COM object
See also Properties and methods, COM object
control-frame COM object 16–4

Properties and methods, COM object
accessing 14–5
data type conversions A–1
See also Data type conversions
default 14–13
locating in viewer 14–21
mode and type options 14–10
restrictions 14–13
syntax 14–5
component handle expression 14–6
data type options 14–9
method name reference 14–7
method reference 14–5
property name reference 14–8
property reference 14–6

Property references, COM object 14–6
restrictions 14–13
_prostar.p procedure 2–4

PROTERMCAP file, colors and fonts 6–2
PSTimer control (OCX) 16–35
PUT statement 5–31, 7–30
PUT-BYTE statement 1–64
PUT-datatype statements 8–5
PUT-KEY-VALUE statement 6–21
example 6–21

Q

Queries 1–19
See also Fetching records
field lists 1–30, 1–33
results lists 1–23

QUIT option
SYSTEM-HELP statement 10–13

Quitting help from application 10–21

QUOTER utility
example procedure D–4
interactive D–6
options
columns in fields D–6
field delimiter D–5

R

RAW data type 1–64

Raw database events
auditing 3–8

Raw disk I/O, UNIX systems 18–23

RAW statement 1–64

Raw terminal I/O mode 18–23
UNIX system recovery 18–25

Reading multiple tables. See Joins

RECID data type
See also ROWID data type
converting to ROWID 1–22
DBRESTRICTIONS function 1–22

Record buffers 1–15

Record phrase
field lists 1–30

Records
adding 1–17
adding to a permissions table 2–42
deleting 1–17
fetching 1–19

Redirecting I/O 5–24
input streams 5–13, 5–16
output streams 5–3, 5–7

REGISTER-DOMAIN method 2–24

Registry
changing resources 6–21
colors and fonts 6–2
Registry flags, Automation objects 15–3

RELEASE OBJECT statement
ActiveX controls 16–18
Automation objects 15–8
COM objects, in general 14–16

RELEASE statement 1–15

Releasing COM objects 14–16

REMOVE-EVENTS-PROCEDURE method 16–24

Reporting
audit 3–19

Reports 7–2
aggregate values 7–12
control blocks 7–3
editors 7–10
multiple destinations 7–23
multiple-table 7–15
output statements 7–3
printing 7–22, 7–34
sending to other devices 7–20
text widgets 7–3
viewing on screen 7–6

REPOSITION statement 1–26
ROWID 1–22

Resource, definitions, changing 6–21

Results lists 1–23
navigating 1–25

Retrieving records. See Fetching records

Return Codes, DDE 13–9

RETURN PARAMETER option
parameter definitions 12–6
RUN parameter lists 12–19

RGB-VALUE function 6–2, 6–7
ActiveX controls 6–7

RGB-VALUE function, COM objects 14–15

ROLES attribute 2–13
using with CAN-DO function 2–42

ROWID data type 1–20
converting from RECID 1–22
CREATE statement 1–22

DataServer
portability 1–22
value construction 1–22
versions 1–22
views 1–22

INSERT statement 1–22
RECID 1–21

REPOSITION statement 1–22
returning record values 1–21
storing values 1–21
value scope 1–22

ROWID function 1–21

Run mode, ActiveX control 8–22
requirements 8–23

RUN statement for DLLs
options 12–19
parameter data types 12–20

Run-time authorization 2–2, 2–39

Run-time connection, methods list 1–5

Run-time OCX properties 16–7

SAVE CACHE statement 1–69

Schema authorization
defined 2–2

Schema Cache File (-cache) startup
parameter 1–72

Schema cache files
building 1–69
creating 1–69
data example 1–71
DataServers 1–69
described 1–69
startup 1–72

Schema definition 10–3

Screen
buffers 1–15

Screen display, HLC library functions C–4

SCREEN-IO option 5–9

Scrolling queries 1–23

SEAL method 2–14

SEAL-TIMESTAMP attribute 2–11

SEARCH function 5–7

Search list, OCX event procedures 16–24

Secure database connections 2–37

Secure Sockets Layer (SSL)
database connections 2–37
socket connections 17–18
Security
activity authorization procedures 2–43
application 2–1
authentication 2–2
authorization, activities-based 2–42
authorizing user IDs at run-time 2–39
compile-time authorization
defined 2–2
connection authorization
defined 2–2
creating, permissions table 2–42
cryptography 2–51
defined 2–2
See also Cryptography
OpenEdge internal
password 2–4
user ID 2–4
password
validating OpenEdge internal 2–4
permissions table 2–47
run-time authorization
defined 2–2
schema authorization
defined 2–2
using secure database connections 2–37
SECURITY-POLICY system handle 2–9
application trusted domain registry 2–23
cryptography 2–52
OpenEdge session ID setting 2–9
SELECT mouse button 4–22
SELF system handle, ActiveX control
events 16–23
Sequence definitions
creating and maintaining 1–57
storage 1–57
Sequence statements and functions 1–58
_Sequence table 1–57
Sequences
accessing values 1–58
attributes 1–54
compiling procedures 1–62
control tables 1–54
performance 1–56
storage limits 1–57
trade-offs 1–54
transaction handling 1–56
defined 1–53
dumping and loading 1–58
fields 1–54
security 1–57
storage 1–57
transactions 1–56
WHERE clause 1–62
Servers, Automation
accessing 15–3
defined 15–1
SESSION-ID attribute 2–11
-sesstimeout connection parameter
socket servers 17–20
SET statement 10–4, 1–15
SET-APPL-CONTEXT method 3–9
usage 3–14
SET-CLIENT method 2–9
SET-CONTENTS option
SYSTEM-HELP statement 10–13
SET-DB-CLIENT function 2–8
SET-PROPERTY method 2–13
SET-SIZE statement 8–3
freeing DLL-allocated memory 12–17
freeing memory 8–4
SETUSERID function 2–5, 2–6, 2–8
Shared buffer access, HLC library functions
C–3
Shared libraries
data types 12–9
loading and unloading 12–22
RAW data type 12–16
routine execution 12–19
structure parameters 12–16
using 12–2
Shared library (UNIX), parameter definition
12–5
Shared library structures, incorrect
initialization 12–16
Shared library, defined 12–1
Shared object, defined 12–1
Shared queries, field lists 1–31
Shared Variable Access, HLC library
functions C–2
SHORT data type 12–10
running DLL routines 12–20
SHORT type specifier 14–10
Single-item transfers, clipboard 9–4
defined 9–2
example 9–8
implementing 9–6
techniques 9–4

Index

Sizing ActiveX controls 16–10

Socket programming 17–1
  examples 17–23
  overview 8–25
Secure Sockets Layer (SSL) 17–18
  client connection parameters 17–21
  configuring servers and clients 17–19
  connecting SSL clients 17–21
  enabling SSL server connections 17–20
  server connection parameters 17–20
  socket operations 17–21

Sockets
  ABL objects 8–27
  connection model 8–26
  defined 8–25
  event model 8–27
  programming requirements 8–29
See also Socket programming

SSL
  See Secure Sockets Layer (SSL)

-ssl connection parameter
  database clients 2–38
  socket clients 17–21
  socket servers 17–20

Startup parameters
  connection 1–2
  Field List Disable (-fldisable) 1–33
  grouping 1–2
  multi-database connection 1–2
  Number of Databases (-h) 1–2
  Physical Database Name (-db) 1–2
  Schema Cache File (-cache) 1–72
  Startup Procedure (-p) 1–6
  Time Stamp (-tstamp) 1–62
  Trigger Location (-trig) 1–63

Startup procedure
  specifying 1–6

Startup Procedure (-p) startup parameter 1–6

STATE-DETAIL attribute 2–11

State-free identity management 2–33

Stateless identity management 2–31

Statements
  data handling 1–15

Static data 18–15

Status area 10–2

STATUS-AREA attribute 10–2

STDCALL option, PROCEDURE statement 12–6

Stream I/O 5–9
  converting widgets 5–9

STREAM-IO option 7–9

Streams 5–2, 5–21
  assignment 5–2
  closing 5–8
  default 5–2
  defining additional 5–17
  establishing 5–17
  input 5–2, 5–17
  opening 5–18
  output 5–2, 5–17
  redirecting 5–13
  See also OUTPUT statements
  redirecting input. See OUTPUT statements
  shared 5–20
  unnamed 5–2

Streams, two-way 5–23

String conversion, ROWID data type 1–21

STRING function, ROWID 1–21

SUBSTRING function
  QUoter utility D–5

Symmetric cryptography 2–56
  See also Cryptography

SYMMETRIC-ENCRYPTION-ALGORITHM
  HM attribute 2–52

SYMMETRIC-ENCRYPTION-IV attribute
  2–53

SYMMETRIC-ENCRYPTION-KEY attribute 2–52

SYMMETRIC-SUPPORT attribute 2–53

Syntax for COM objects 14–5

System clipboard
  CLIPBOARD handle
    See also CLIPBOARD system handle
  data transfer modes 9–2
  defined 8–12, 9–1
  enabling and disabling operations 9–4
  example procedure
    multiple-item transfer 9–14
    single-item transfer 9–8
  FOCUS handle 9–4
  handle attributes 9–2
  multiple-item transfers 9–12
  data-based 9–13
  defined 9–2
  widget-based 9–12
operations
copy 9–7
cut 9–6
defined 9–2
paste 9–6
single-item transfer
defined 9–2
element 9–8
implementing 9–6
techniques 9–4

System handles, ActiveX control events
16–23

SYSTEM-DIALOG COLOR statement
6–13
element 6–15

SYSTEM-DIALOG FONT statement 6–13
example 6–17

SYSTEM-DIALOG PRINTER-SETUP
statement 5–4, 5–5

SYSTEM-HELP statement 10–9
QUIT parameter 10–21

T

Tab order of ActiveX controls
design time 16–11
run time 16–16

Tables
audit policy 3–20
joining. See Joins
reading multiple 1–34
word break 1–41

Tables, audit data 3–22

Temporary tables
joins 1–35

Terminal I/O, UNIX Systems 18–23

Terminating sequences, resetting 1–62

TEXT option
SYSTEM-HELP statement 10–14

Tilde (~), input files 5–33

Time Stamp (-tstamp) startup parameter
1–62

Timer services
defined 18–19
HLC library functions C–4

ToolTip 10–6
defining in the AppBuilder 10–7

Topic IDs 10–10

Top-level Automation objects 15–2

TO-ROWID function 1–21

Transactions 18–20

Trigger Location (-trig) startup parameter
1–63

Triggers
help 10–18, 10–20

Trust domain registries 2–16

Two-way streams 5–23

TYPE attribute 9–3

Type Libraries, defined 14–10

Type options
purpose 14–10
using 14–11

Type specifiers
COM object syntax 14–9
purpose 14–10
using 14–11

U

Universal unique identifier
creating 2–36

UNIX
compiling source files 18–26
HLC applications 18–23
shared libraries
and ABL 8–14
and HLC 18–3
defined 8–14

UNIX, processes as input and output
streams 5–23

UNLOAD statement
using 6–23

Unnamed streams 5–2

UNSIGNED SHORT data type 12–10
running DLL routines 12–20

UNSIGNED-BYTE type specifier 14–10

UPDATE statement 1–15, 10–4
USE statement using 6–23

User count, exceeding 1–2

User identities 2–7
  See also Identity management

User IDs
  authorizing at run time 2–39
  CAN-DO validation list 2–41
  OpenEdge internal
    blank 2–5
    rules for forming 2–4
    validating 2–4

User interrupt handling 18–20

USER-ID attribute 2–12

USERID function 2–41

User-interface triggers 4–28
  instead of EDITING blocks 4–30

V

VALIDATE-SEAL method 2–15

Validating component handles 14–14

Validation
  field-level 5–29

VALUE attribute 9–3
  multiple-item transfers
    example 9–14
  single-item transfers 9–6
    example 9–8

VALUE option
  OUTPUT TO statement 5–8

Version, of OpenEdge, determining the current 4–16

VIEW-AS phrase 10–7

Visual colors, Windows color dialog 6–14

W

Widget handles, compared to component handles 14–2

Widget hierarchy 10–20

Widgets
  converting for stream I/O 5–9

Window status area 10–2

WINDOW-NAME option
  SYSTEM-HELP statement 10–11

Windows
  clipboard output 5–6
  color 6–11
    Palette Manager 6–11
  printer setup 5–4, 5–5

Windows NT named pipe 11–14

WinExec() DLL function
  code example 13–9
  DDE 13–9

Word break tables 1–41
  internationalization 1–41

Word indexing 1–41

Word-break, rules file
  applying to database 1–47
  compiling 1–47

Work tables
  joins 1–35