Progress®



# Troubleshoot ABL Applications

OpenEdge®

# Copyright

© **2019 Progress Software Corporation and/or its subsidiaries or affiliates. All rights reserved.**

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

Corticon, DataDirect (and design), DataDirect Cloud, DataDirect Connect, DataDirect Connect64, DataDirect XML Converters, DataDirect XQuery, DataRPM, Deliver More Than Expected, Icenium, Kendo UI, Kinvey, NativeScript, OpenEdge, Powered by Progress, Progress, Progress Software Developers Network, Rollbase, SequeLink, Sitefinity (and Design), Sitefinity, SpeedScript, Stylus Studio, TeamPulse, Telerik, Telerik (and Design), Test Studio, and WebSpeed are registered trademarks of Progress Software Corporation or one of its affiliates or subsidiaries in the U.S. and/or other countries. Analytics360, AppServer, BusinessEdge, DataDirect Autonomous REST Connector, DataDirect Spy, SupportLink, DevCraft, Fiddler, JustAssembly, JustDecompile, JustMock, Kinvey Chat, NativeChat, NativeScript Sidekick, OpenAccess, ProDataSet, Progress Results, Progress Software, ProVision, PSE Pro, SmartBrowser, SmartComponent, SmartDataBrowser, SmartDataObjects, SmartDataView, SmartDialog, SmartFolder, SmartFrame, SmartObjects, SmartPanel, SmartQuery, SmartViewer, SmartWindow, and WebClient are trademarks or service marks of Progress Software Corporation and/or its subsidiaries or affiliates in the U.S. and other countries. Java is a registered trademark of Oracle and/or its affiliates. Any other marks contained herein may be trademarks of their respective owners.

**February 2019**

**Last updated with new content:** Release 12.0

**Updated: 2019/03/01**

# Table of Contents

**Preface....11**
Purpose....11
Audience....11
Organization....12
Using ABL documentation....12
References to ABL compiler and run-time features....12
References to ABL data types....13
Typographical conventions....13
Examples of syntax descriptions....14
Long syntax descriptions split across lines....16
Complex syntax descriptions with both required and optional elements....16
OpenEdge messages....16
Obtaining more information about OpenEdge messages....17

**Part I: OpenEdge Debugger....19**

**Chapter 1: Introduction....21**
Debugger system requirements....21
Debugger features and functions....22
Application control....22
Application data management....23
Execution monitoring....24
Session management and information....24
Enabling debugging....25
Understanding debugging sessions....26
Interaction between Debugger and client....26
Debugger and application control....27
Procedures and the Debugger context....27
The Debugger context....27
Debugger context and the procedure call stack....29
Debugging in a distributed environment....29
Distributed debugging....30
Remote debugging....30
Compatibility issues....30

**Chapter 2: Starting and Stopping the Debugger....33**
Debugging modes....33

Application mode....34
Stand-alone mode....35
Attachable mode....35
Making the process ready for debugging....36
Stacked applications....39
Debugging from the Procedure Editor....39
Debugging from the AppBuilder....40
Debugging from the ADE Desktop....40
Debugging from the OpenEdge command line....41
Debugging from alert dialog boxes....42
Displaying the .NET Stack Trace in the Stack Trace dialog box....42
Displaying the ABL Error CallStack in the Stack Trace dialog box....43
Starting the Debugger from the Stack Trace dialog box....45
Debugging from an ABL procedure....45
Debugging in attachable mode....46
Detaching from the process....46

**Chapter 3: Debugger Window and Files....47**

Debugger window layout....48
Debugger window panes....48
Source code pane....49
ABL stack trace pane....50
Variables pane....50
Watches pane....53
Debugger title bar....56
Debugger menu bar options....56
Accelerator keys and mnemonics....56
File menu....56
Edit menu....57
Search menu....57
View menu....57
Debug menu....58
Diagnostics menu....59
Data menu....60
Window menu....60
Help menu....60
Debugger toolbar buttons....60
Debugger status bar....62
Debugger dialog boxes....62
Add Watch dialog box....62
Attach to Process dialog box....63
Breakpoints dialog box....64
Dataview dialog box....65
Dynamic Object Tracking dialog box....70

Edit Breakpoint dialog box ....76
Find dialog box....77
New Breakpoint dialog box....77
Open dialog box....83
Preferences dialog box....83
Select Process dialog box ....86
Debugger files....86
Debug listing files ....86
Locating debug listing files in attachable mode....87
Procedure files and listing generation ....87

**Chapter 4: Using System Handles for Debugging....89**

DEBUGGER system handle attributes and methods ....90
Using the DEBUGGER system handle to start and control the Debugger....91
Using the DEBUGGER system handle in application mode ....91
Using the DEBUGGER system handle in stand-alone mode....93
Using the SESSION handle to start the Debugger....95

**Part II: Troubleshooting Resources....97**

**Chapter 5: Logging in OpenEdge....99**

Logging infrastructure overview....99
LOG-MANAGER system handle attributes and methods ....100
DSLOG-MANAGER system handle attributes and methods....102
Log files....104
Specifying the log filename....105
Clearing the log file....106
Writing user messages to the log file....106
Closing the log file....107
Log size threshold....107
Specifying the log size threshold....108
Specifying the number of log files retained....108
Filename for rolled-over log files....109
srvrLogWatchDogInterval property ....110
Log entry types....110
Log entry types (Client context)....110
Log entry types (Server context)....115
Specifying log entry types....116
Logging levels....117
Specifying the logging level....118
Logging settings ....119

## Chapter 6: Log Entry Types Detail.....121

ABL message logging.....122
ABL trace logging.....122
ABL transaction logging.....124
AIA logging.....126
AppServer logging.....127
Database connection logging.....128
DataServer logging.....128
DataServer client context logging.....128
DataServer server context logging.....129
DataServer logging switch for ABL to SQL correlation.....130
Dynamic object logging.....131
Logging dynamic object creation and deletion .....134
Deleted objects .....135
Objects created and deleted implicitly.....135
Logging levels for dynamic objects.....135
Event logging.....139
Event generation.....139
Event stack .....139
Logged events.....139
Logged information.....140
Event log entry types and logging levels.....142
Starting event logging.....143
Event message formats .....144
Sample Event logs.....148
File access logging.....149
Ignored operation logging.....150
NameServer logging.....151
Query information logging.....152
Query information log entry type and logging levels.....152
Starting query information logging.....152
Generating query information.....154
Query information logged.....156
Query plan.....156
Query statistics.....157
Sample query information logs .....162
SAX parser logging .....165
Temp-table logging.....166
Unified Broker logging.....167
Web Services Adapter (WSA) and Web Service logging.....168
WSA Log entry types.....168
Setting the WSADefault log entry type.....171
Setting the loggingLevel property.....172

Setting the serviceLoggingLevel property....172
Setting the serviceFaultLevel property....172
Understanding Web service information in WSA logs....173
WebSpeed Messenger logging....173

**Chapter 7: Other Diagnostics Resources....175**

Abnormal termination dump files....175
procore file....176
protrace file....176
core file....177
How the AVM handles failure conditions....177
Run-time diagnostics....178
Collecting r-code usage statistics....179
Collecting temp-table usage statistics....181
Supported virtual system tables for temp-tables....182
ABL classes for accessing temp-tables....183
Index Statistics file....183
Table Statistics file....184
Progress.Database.TempTableInfo:GetTableStatHistoryHandle()....185
Analyzing ABL code....187
Generating a protrace file for a running program....188
Analyzing procedures and queries....188
COMPILE XREF option....188
INDEX-INFORMATION attribute of the query object handle....189

**Index....191**

# Preface

For details, see the following topics:


- Purpose
- Audience
- Organization
- Using ABL documentation
- Typographical conventions
- Examples of syntax descriptions
- OpenEdge messages


## Purpose

This book describes the OpenEdge® Debugger and how to use it to find logic errors in ABL (Advanced Business Language) applications. This is both a guide and reference that provides complete information on the Debugger and techniques for using it to monitor and control the execution of ABL procedures.

## Audience

The primary audiences for this book are business application developers and OpenEdge system administrators. The secondary audiences are OpenEdge developers and technical support engineers.

# Organization

Part I describes how to start, stop, and interact with the Debugger as you debug your code. It is a complete guide to the features, terms, and principles of operation you need to know to use the Debugger effectively. Once you have begun a debugging session, the Debugger online help available from the Help menu of the Debugger window provides most of the information you need to use the Debugger.

Part II describes the logging infrastructure and various types of logging available to troubleshoot your code. It also provides guidelines and direction for using other diagnostic tools available for troubleshooting purposes.

Introduction on page 21

Describes the features of the Debugger and how it interacts with an ABL application.

Starting and Stopping the Debugger on page 33

Describes how to start and end a debugging session in all supported operating environments.

Debugger Window and Files on page 47

Describes the Debugger user interface and files created or referenced by the Debugger.

Using System Handles for Debugging on page 89

Describes how to use the `DEBUGGER` and `SESSION` handles, which are ABL system handles that allow you to start the Debugger and control some Debugger features from an procedure.

Logging in OpenEdge on page 99

Describes the enhanced logging infrastructure.

Log Entry Types Detail on page 121

Describes the various types of logging that enhanced logging supports.

Other Diagnostics Resources on page 175

Describes other diagnostics tools and aids available to developers.

# Using ABL documentation

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 go to https://docs.progress.com.

## 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.

# Typographical conventions

This documentation uses the following typographical and syntax conventions:

| Convention | Description |
|---|---|
| **Bold** | Bold typeface indicates commands or characters the user types, provides emphasis, or the names of user interface elements. |
| *Italic* | Italic typeface indicates the title of a document, or signifies new terms. |
| **SMALL, BOLD CAPITAL LETTERS** | Small, bold capital letters indicate OpenEdge key functions and generic keyboard keys; for example, **GET** and **CTRL**. |
| **KEY1+KEY2** | A plus sign between key names indicates a **simultaneous** key sequence: you press and hold down the first key while pressing the second key. For example, **CTRL+X**. |
| **KEY1 KEY2** | A space between key names indicates a **sequential** key sequence: you press and release the first key, then press another key. For example, **ESCAPE H**. |
| **Syntax:** | |
| `Fixed width` | A fixed-width font is used in syntax, code examples, system output, and file names. |

| Convention | Description |
|---|---|
| *Fixed-width italics* | Fixed-width italics indicate variables in syntax. |
| ***Fixed-width bold*** | Fixed-width bold italic indicates variables in syntax with special emphasis. |
| UPPERCASE fixed width | ABL keywords in syntax and code examples are almost always shown in upper case. Although shown in uppercase, you can type ABL keywords in either uppercase or lowercase in a procedure or class. |
| 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. |
| **\|** | A vertical bar indicates a choice. |
| **. . .** | Ellipses indicate repetition: you can choose one or more of the preceding items. |

# 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 ... ] }
```

# 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**.

# I

# OpenEdge Debugger

For details, see the following topics:

- Introduction
- Starting and Stopping the Debugger
- Debugger Window and Files
- Using System Handles for Debugging

# 1

# Introduction

The OpenEdge® Debugger is an interactive tool that allows you to control and monitor the execution of compilable ABL (Advanced Business Language) procedures. With the Debugger, you can execute a procedure and its subprocedures, stepping through procedures line by line or stopping at specified lines and then resuming execution from those points. You can also examine data from the procedure you are running, including the names and contents of variables it defines, and you can modify this data.

In addition, you can use two debugging modes, distributed debugging and remote debugging, to examine code associated with a distributed application.

For details, see the following topics:

- Debugger system requirements
- Debugger features and functions
- Enabling debugging
- Understanding debugging sessions
- Procedures and the Debugger context
- Debugging in a distributed environment

## Debugger system requirements

The Debugger runs in Windows and on UNIX. The Windows version of the Debugger can debug either GUI or character-mode applications. The UNIX version of the Debugger can debug only character-mode applications.

The Debugger is a Java™ application. As such, it requires that the Java Run-time Environment (JRE) component be installed on the same machine. The JRE is the run-time part of Java and consists of the Java Virtual Machine, the Java Core Classes, and supporting files. Depending on the platform on which you install the Debugger, the JRE might not have been included in the installation. In Windows, the OpenEdge installation program automatically installs these Java components. On UNIX, the OpenEdge installation program automatically installs these Java components only on certain platforms.

For more information, see *Install OpenEdge*.

# Debugger features and functions

The Debugger is a utility you can run with any OpenEdge application. When you invoke the Debugger, it runs in a separate process with its own window. The Debugger window provides access to all Debugger features and functions through menu options, toolbar buttons, or accelerator keys. The Debugger window also provides access to online help information on all Debugger features and functions. For more information on the Debugger window and how to access all Debugger features and functions, see Debugger Window and Files on page 47

You can invoke the Debugger while running in an OpenEdge session, from the Procedure Editor or AppBuilder, for example. You can run the Debugger in attachable mode; that is, you open the Debugger first and then attach to a running client process. For more information, see Debugging in attachable mode on page 46

You can also invoke the Debugger from an ABL procedure, directly or indirectly, using the `DEBUGGER` and `SESSION` system handles. With these system handles, you can control some Debugger features from an ABL procedure and build your own OpenEdge development tools that include the Debugger as a tool option. For more information, see Using System Handles for Debugging on page 89

The following sections describe the Debugger features and functions:


- Application control on page 22
- Application data management on page 23
- Execution monitoring on page 24
- Session management and information on page 24


## Application control

During a debugging session, the Debugger allows you to start and stop procedure execution as you want, and even execute procedures that are not part of your application. All procedures the Debugger executes are added to the procedure call stack. The *procedure call stack* is a last-in-first-out (LIFO) memory structure that keeps track of the execution status of each procedure as it calls a subprocedure or invokes a trigger block.

The following sections cover:


- Open and incremental execution on page 22
- Procedure interruption on page 23
- Procedure termination on page 23


### Open and incremental execution

To execute a procedure other than the one the Debugger is currently running, open the procedure and start it from the Debugger, using **Debug** > **Run**. Whenever the Debugger has control, you can run any application, effectively adding it to the procedure call stack of your current application. This allows you to mix and match procedures dynamically or run diagnostic procedures while debugging your application.

**Note:** You cannot start a procedure that requires parameters or arguments from the Debugger. The Debugger has no way to define them.

When the Debugger is stopped in an executing procedure, you can continue execution, using **Debug** > **Continue**. This allows the procedure to execute until it encounters a breakpoint, an interrupt, or the end of the procedure. You can also continue executing the current procedure line by line using **Debug** > **Step Into Debug** > **Step Out** , or **Debug** > **Step Over**. Use **Debug** > **Start Animation**to watch your program execute as the Debugger steps through your code. Use the **Run to Line** command to continue program execution up to a specified line of ABL code.

### Procedure interruption

To selectively interrupt procedure execution, you can set or remove breakpoints, using **Edit** > **Breakpoints**. A *breakpoint* can be a specific line in a procedure where execution stops temporarily. When the procedure reaches a breakpoint, it halts before executing the specified line and gives control to the Debugger for your next command. The easiest way to set a breakpoint is to click in the left column of the source code pane where the procedure appears.

A breakpoint can be conditional, where evaluation of a logical expression determines whether the interruption occurs. A breakpoint can be based on encountering an error, in which case the Debugger gets control at the next executable line after the line that caused the error.

You can set and remove breakpoints in an ABL procedure, using the `DEBUGGER` system handle's `SET-BREAK( )` and `CANCEL-BREAK( )` methods. This feature is helpful to control breakpoints at specific points in your code, regardless of how the surrounding code is modified, or to set breakpoints in include files that might appear in many different procedures.

To interrupt a running procedure and give control to the Debugger, the Debugger provides a **Debug** > **Interrupt** command. This interrupts a procedure wherever it is executing, as if it encountered a breakpoint. It is particularly useful when you want to interrupt a procedure in which you have not set a breakpoint.

**Note:** If you use **Debug** > **Interrupt** while an alert dialog box is displayed on the screen, or while the procedure is executing a `READKEY` or `PAUSE` statement, the Debugger does not take control until you dismiss the alert dialog box or satisfy the `READKEY` or `PAUSE` statement.

You can provide a means to interrupt a running ABL procedure by setting the `SESSION` system handle `DEBUG-ALERT` attribute to `TRUE`. This places a **Help** button on all alert boxes that provides access to a **Stack Trace** dialog box and the Debugger.

### Procedure termination

You can force a procedure to terminate at a break or interrupt point, using **Debug** > **Stop**. This effectively inserts and executes an ABL `STOP` statement at the current break or interrupt point.

## Application data management

Whenever the Debugger has control, you can monitor and make changes to data items in your application. For example, you can:

- Examine the values of variables, buffers, parameters, temp-tables, and datasets that are defined and accessible from an executing procedure in the variables pane. You can modify the values of variables and parameters.
- Examine and modify the values of attributes for objects, using **Dataview** > **Debug** .
- Examine and modify the values of database and temp-table buffer fields, using **Debug** > **Dataview**.

- Examine and modify the elements of an array selected in the variables pane, or of an array field selected in a buffer, using **Debug** > **Dataview**.
- Add watches to monitor the values of any variables, attributes, fields, or built-in ABL functions in the executing procedure, or any procedure on the call stack, using **Debug** > **Add Watch**.

## Execution monitoring

The Debugger provides several ways to examine the state of an application, as described in the following sections:


- Code path analysis on page 24
- Transaction monitoring on page 24
- Breakpoint analysis on page 24
- Dynamic object monitoring on page 24


### Code path analysis

During procedure execution, the Debugger window displays the listing of the procedure at each breakpoint or stopping point in its execution. A pointer always indicates the next line to execute. At any stopping point, you can display the listing for any procedure file using **File** > **Open**. This provides a convenient way to set breakpoints for any procedure in your environment, whether or not it is called from the procedure you are running.

You can examine and navigate the procedure call stack using the ABL stack trace pane. This allows you to examine the current code path back and forth through the active series of procedure calls and triggers starting from the current stopping point.

### Transaction monitoring

You can determine the transaction state of your application using **Debug** > **Show Transaction**. This tells you whether a transaction is active at the current stopping point.

### Breakpoint analysis

You can display the current breakpoint settings using **Breakpoints** > **Edit**. This lists the breakpoints currently set for all procedures during the session. The listing for each executing procedure also indicates each line where a breakpoint is set.

### Dynamic object monitoring

You can monitor the creation and destruction of dynamic object instances in your application, using **Diagnostics** > **Monitor Dynamic Objects**. This lets you track dynamic object instances your application has created and not yet destroyed. This can help you locate unused object instances that can result in memory leaks.

## Session management and information

The Debugger provides several features that allow you to monitor and customize a debugging session. These features include:


- Window settings on page 25
- Listing management on page 25

- ABL Debugger control on page 25

### Window settings

The Debugger automatically saves some window settings, such as window size and location, across debugging sessions. You can set user preferences for other Debugger window settings, such as text font and size, using **Edit** > **Preferences**.

### Listing management

The Debugger window provides a search menu that allows you to search forward and backward for text in the displayed procedure listing.

### ABL Debugger control

The `DEBUGGER` and `SESSION` system handles provide attributes and methods that allow you to initialize, start, and control features of a debugging session directly or indirectly from within an ABL procedure. For example, you can use these handles to display and remove the Debugger window from the screen and selectively interrupt procedure execution.

## Enabling debugging

Prior to running the Debugger on an OpenEdge installation that contains an ABL Virtual Machine (AVM), you must enable the system for debugging. Attempts to run the Debugger without first enabling debugging result in an error message.

Debugging is initially disabled because it opens a potential security hole. When running the Debugger, you can examine and modify internal variables and database fields. In a production environment, this means that someone could access sensitive business data while in the Debugger. For this reason, Progress Software Corporation strongly recommends that you avoid using the Debugger on a production system.

During application development, however, you can choose whether to selectively enable and disable debugging or enable it after installation and leave it enabled for the duration of the development cycle.

To enable debugging, use the `proDebugEnable` command.

This is the syntax for the command:

```
proDebugEnable { -disable-all | -enable-all }
```

`disable-all`

Disables all AVM instances that are part of the installation from which you are executing the command. Specify this parameter to disable debugging on an installation where it was previously enabled.

`enable-all`

Enables all AVM instances that are part of the installation from which you are executing the command

To further secure the debugging facility, you must have the right privileges to execute this command: root privileges on UNIX and administrator privileges in Windows.

Note the following about enabling debugging:

- The proDebugEnable command controls debugging of AVM instances running on the machine where you execute it. It does not affect running of the Debugger user interface. So, for example, if you want to run the Debugger on one machine to debug an AVM instance that is on another machine (across a LAN), you must run proDebugEnable on the machine where the AVM instance is running.
- If you install OpenEdge on machine 1 and do a network install onto machine 2, you must run `proDebugEnable` on machine 2 to debug an AVM instance that is running on machine 2.
- If you install multiple instances of OpenEdge on the same machine, each instance will have its own copy of `proDebugEnable`, and each copy will affect only the debugging capability in its instance.

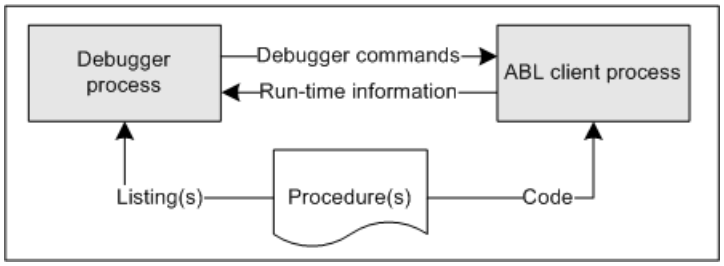
  Thus, you can run an application system and a production system on the same machine, and control debugging separately for each system.
- If you anticipate that you might want to use the Debugger, execute proDebugEnable before starting the AVM. If you decide you want to debug after starting the AVM, you will have to shut down the AVM, enable debugging, and then restart the AVM.
- If you want to debug an AppServer™ or WebSpeed® agent, you also must set the appropriate `debuggerEnabled` property to `1` in the `ubroker.properties` file, in addition to enabling debugging with the proDebugEnable command.

**Note:** The default value for the `debuggerEnabled` property is `0`.

# Understanding debugging sessions

During a debugging session, the Debugger user interface and ABL client processes cooperate to execute, monitor, and interrupt application procedures. The ABL client process executes your application procedures, and the Debugger process manages the Debugger window. However, the two processes share the management of application execution. The following figure shows this interaction.

**Figure 1: Debugger/client interaction**




## Interaction between Debugger and client

The Debugger process sends commands to the client based on menu options you select and other actions you take in the Debugger window (such as setting breakpoints). The client interprets the commands, performs relevant actions (such as honoring breakpoints), and sends run-time information to the Debugger process in response. For example, the client sends the procedure and line number of each breakpoint it encounters. The Debugger process uses the run-time information it receives from the client to manage and display procedure listings, variable values, and so on.

## Debugger and application control

Either the Debugger or the executing application has control during a debugging session. The Debugger has control when no application is running or the running application is interrupted, for example, at a breakpoint. When the Debugger has control, it can send commands to the client process for execution. The application has control whenever it is executing ABL code, for example, in response to a Debugger control flow command such as **Debug** > **Continue**. When the application has control, it interacts with the user and database as if the Debugger were not running.

The following sections detail the Debugger and application interaction:


- Application interaction during Debugger control on page 27
- Debugger interaction during application control on page 27


### Application interaction during Debugger control

When the Debugger has control, the application is blocked. At this point, the client application is waiting to execute a line of code. The application continues to wait until you invoke a Debugger control flow command such as **Debug** > **Step Into**, **Debug** > **Step Over**, **Debug** > **Step Out**, or **Debug** > **Continue**.

### Debugger interaction during application control

When the application has control (is executing), you cannot invoke commands in the Debugger window until the Debugger receives control (for example, when the application reaches a breakpoint). You can force the Debugger to take control during procedure execution by using **Debug** > **Interrupt**.

# Procedures and the Debugger context

This section introduces several terms that are used throughout this manual. These terms describe the relationship between the Debugger and application procedures it controls or references, and include:


- The Debugger context on page 27
- Debugger context and the procedure call stack on page 29


## The Debugger context

The *Debugger context* consists of the data and execution status of all procedures running under Debugger control.

The following sections describe:

- Procedures and triggers
- Persistent procedures
- Callback procedures
- Data
- Execution status

## Procedures and triggers

Procedures and trigger blocks in the Debugger context include the executing procedure or trigger plus some or all of the procedures and triggers pushed onto the procedure call stack. The number of procedures and triggers on the call stack increases as each subprocedure or trigger calls another and decreases as each subprocedure or trigger returns to the one that called it. The Debugger provides information on a running application by referencing this call stack.

The executing procedure and all procedures on the call stack (whether inside or outside the Debugger context) are *active procedures*. Active procedures include all those that you run directly or indirectly from the client and any additional procedures you start from the Debugger with **Debug** > **Run**.

*Inactive procedures* are procedures for which the Debugger cannot examine data. They include:

- Procedures that have yet to be called
- Procedures that have completed execution and have returned to their caller
- Instances of persistent procedures that have been deleted from the session
- Instances of persistent procedures that are not currently executing

## Persistent procedures

A persistent procedure is active in the Debugger context in the following instances:

- When the persistent procedure is first executing to instantiate it (`RUN ... PERSISTENT`)
- When one of its triggers is executing in response to a user or program event
- When one of its internal procedures is executing from another procedure (`RUN ... IN`)

Otherwise, the persistent procedure is not on the call stack and is invisible to the Debugger context.

## Callback procedures

A *callback procedure* is an internal procedure that can be set by an ABL program; for example, when generating or receiving a Web Services Out SOAP message, or when parsing an XML document. You can use the Debugger to set breakpoints in callback procedures.

## Data

Data in the Debugger context includes all variables, buffers, parameters, temp-tables, and attributes and fields of variables and objects defined or accessed by procedures inside the Debugger context.

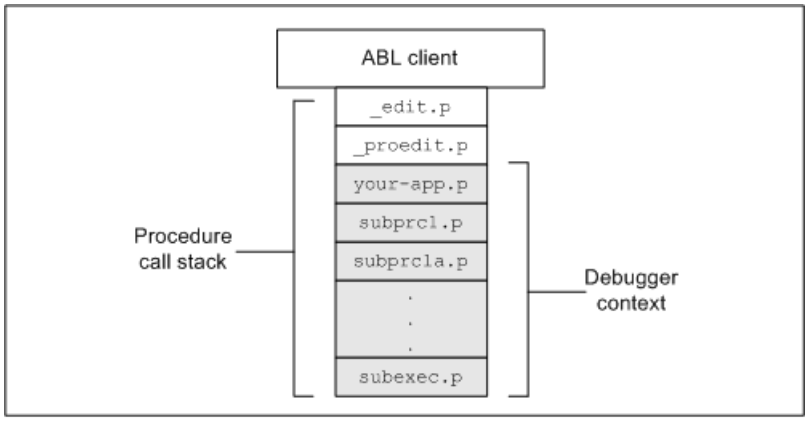
## Execution status

The execution status includes the location of the current *executable line* and the relative position in the call stack of each procedure in the Debugger context, plus the current transaction state. An executable line is any ABL line that contains a run-time statement (for example, `FOR`, `DISPLAY`, or `CREATE`). Although the listing monitored by the Debugger contains all the ABL code for a procedure, the execution status reflects only executable lines; execution stops only on executable lines. *Nonexecutable lines* include comments and static data definition statements (for example, `DEFINE` or `FORM`). Static data definition statements take effect only at compile time and generate data items and objects for a procedure.

## Debugger context and the procedure call stack

The following figure shows the relationship between active procedures inside and outside the Debugger context in a typical procedure call stack (growing downward in the figure). The Debugger is controlling an application (`your-app.p`) that is running in the Procedure Editor (also an OpenEdge application). In this figure, the executing procedure is `subexec.p`. The procedure call stack includes all procedures down to the executing procedure. All procedures running from your application are in the Debugger context. Only procedures associated with the Procedure Editor (`_edit.p` and `_proedit.p`) are running outside of the Debugger context.

**Figure 2: Procedures inside and outside the Debugger context**




An active procedure can run outside the Debugger context in one of the following two ways:

- When the name of the procedure is preceded by an underscore (_), as with `_edit.p` and `_proedit.p`
- When the procedure invokes the Debugger with the `DEBUG` method of the `DEBUGGER` system handle

For more information on using underscores in procedure names, see Debugger Window and Files on page 47. For more information on the `DEBUGGER` system handle, see DEBUGGER system handle attributes and methods on page 90.

The Debugger keeps track of run-time information only for active procedures inside the Debugger context. The Debugger allows you to examine the data items, object attributes, listing, and execution status of any procedure that is active in the Debugger context anytime the Debugger has control.

You can neither examine data from nor control execution of procedures outside the Debugger context.

# Debugging in a distributed environment

The OpenEdge Debugger supports debugging application code in a distributed environment in two modes, distributed debugging and remote debugging. These modes provide two different models for debugging your distributed application. Both modes use the same Debugger window to debug remote procedures.

---

**Note:** Running the Debugger in attachable mode can be considered a form of debugging in a distributed environment, as you can run the Debugger on one machine and debug a client process on another machine within the same LAN. For more information, see Debugging in attachable mode on page 46

---

The following sections describe debugging in a distributed environment:

- Distributed debugging
- Remote debugging
- Compatibility issues

## Distributed debugging

*Distributed debugging* is the ability to debug associated ABL processes within a distributed application environment. That is, one debugger session can control both the client process and AppServer process, giving the client session the ability to step into code running on the AppServer. The procedure call stack reflects all procedures in all processes involved, enabling the root client application to be aware of the AppServer processes it is debugging. The *root client application* is the client application that initiates the Debugger using distributed debugging.

You can use distributed debugging on AppServers running in any session-managed operating mode (that is, state-reset, state-aware, stateless, but not state-free).

You can run the Debugger using distributed debugging to debug local procedures, remote procedures, and remote procedures executed by other remote procedures. An AppServer process that is once, twice, or further removed from the initial debugger session can be debugged in this manner.

## Remote debugging

R*emote debugging* is the ability to debug server-side and remote application procedures. This mode prevents debugging any procedure in a client application session. Remote debugging allows you to debug application code running on an AppServer process or a WebSpeed agent independently of the calling client application. Remote debugging is useful for debugging remote procedures that are:

- Running on an AppServer
- Running on a WebSpeed agent
- Executed by an Open Client (non-ABL client) application
- Executed by an OpenEdge Web Service
- Executed asynchronously on the AppServer from an ABL client application

In this debugging mode, the client application driving an AppServer process is unaware that the Debugger is running on the AppServer process. On the server side, the procedure call stack shown in the Debugger's ABL stack trace pane reflects only entries up to the top level remote procedure call. It does not reflect entries for the ABL client application that is driving the AppServer process.

## Compatibility issues

Changes in the Debugger over time have lead to certain incompatibilities when debugging in a distributed environment when a client and an AppServer are running on different software releases.

The client and AppServer use a protocol to communicate during debugging. There have been changes to this protocol such that an AppServer running on an earlier or later release cannot communicate effectively with the client in some configurations.

Added features in a later release introduce new commands to support these feature requests. An AppServer running on an earlier release cannot service the new feature requests because it does not recognize the commands introduced in the later release.

The Debugger reconciles release discrepancies sufficiently in some cases to continue executing, and in others, to provide feedback on why it cannot continue. The following table summarizes what happens when attempting client/AppServer debugging with different software releases.

**Table 1: The Debugger and mixed client/AppServer releases**

| Client | AppServer | Results |
|---|---|---|
| 9.1D05/06/07 | 9.1D05/06/07 | The client and AppServer must be running on identical releases, that is, 9.1D05 to 9.1D05, 9.1D06 to 9.1D06, and 9.1D07 to 9.1D07 to debug successfully; otherwise, results are unpredictable. |
| 9.1D08 | 9.1D07 or earlier | Incompatible protocols prevent the client and AppServer from communicating. The Debugger displays an error message to this effect; the procedure executes without debugging. |
| 9.1D08 | 10.0A and later | The client protocol adapts to the protocol of the AppServer for debugging purposes, but cannot take advantage of the features in the later release. |
| 10.0A | 9.1D07 or earlier | Incompatible protocols prevent the client and AppServer from communicating. The Debugger displays an error message to this effect; the procedure executes without debugging. |
| 10.0A and later | 9.1D08 and later | The client protocol adapts to the protocol of the AppServer. Debugging proceeds as expected.When debugging on a 9.1D08 AppServer, feature requests dependent on the later release will not execute. For example, conditional breakpoints fire, without first testing the condition; error breakpoints do not fire at all.Other cases might result in an error message from the AppServer because it does not recognize a command new to the later release. |

# 2

# Starting and Stopping the Debugger

Depending on where you are working in OpenEdge and how you want the Debugger to interact with your ABL (Advanced Business Language) application, you can start the Debugger in different ways. The following sections describe techniques for starting and stopping the Debugger.

For more information on the Debugger window, see Debugger Window and Files on page 47

For details, see the following topics:

- Debugging modes
- Stacked applications
- Debugging from the Procedure Editor
- Debugging from the AppBuilder
- Debugging from the ADE Desktop
- Debugging from the OpenEdge command line
- Debugging from alert dialog boxes
- Debugging from an ABL procedure
- Debugging in attachable mode

## Debugging modes

When you start a debugging session, the Debugger initializes and runs in one of three modes: application mode, stand-alone mode, or attachable mode. When and how the Debugger gets control depends on the mode used to run it.

The following sections describe the debugging modes:

- Application mode
- Stand-alone mode
- Attachable mode

## Application mode

The Debugger runs in *application mode* when you start an initial application by means of the Debugger options of the Procedure Editor and the AppBuilder. This application runs in the Debugger context, and the Debugger takes control when the application stops at an automatic breakpoint set at the first executable line. This gives you a chance to examine your application listings and set breakpoints before continuing execution. You can run other procedures from the Debugger, using **Debug** > **Run**, effectively running them under your initial application.

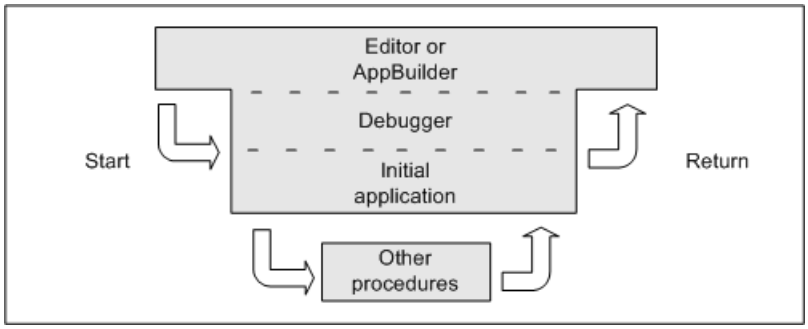
---

**Note:** You can run the Debugger in application mode from your own application, using the `DEBUGGER` system handle. For more information, see Debugging from an ABL procedure on page 45.

---

The following figure shows the relationship between the Debugger in application mode, its initial application, other procedures it might run, and the OpenEdge environment in which it is running.

**Figure 3: Debugging in application mode**




In application mode, you can start and restart the Debugger and its initial application without leaving your OpenEdge environment.

### Exiting procedures started with Debug Run

When you exit a procedure started with **Debug** > **Run**, it returns control to the Debugger. Any previous application on the stack (including the initial application) waits to resume execution at its current stopping point.

### Exiting the Debugger before the initial application

When you exit the Debugger in application mode by closing the Debugger window, the Debugger purges itself from the OpenEdge environment and closes any procedures started with **Debug** > **Run**. It then returns control to the initial application, which continues execution from its current interruption point.

### Exiting the initial application before the Debugger

When you exit the initial application, it returns control to the environment where you started it. If the Debugger is running when you exit, the Debugger closes its window but leaves Debugger session settings, such as window size and location, initialized in the OpenEdge environment. If you then run the same or another application using the Editor or AppBuilder Debugger option, the Debugger window opens with its previous window settings and takes control with a breakpoint at the first executable line.

## Stand-alone mode

The Debugger runs in *stand-alone mode* when you start your OpenEdge session from the command line with the Debugger (`-debug`) startup parameter or from the Application Development Environment (ADE) Desktop. In stand-alone mode, the Debugger takes control immediately as your initial user interface and with no initial application running in the Debugger context.

---

**Note:** You can run the Debugger in stand-alone mode from your own application, using the `DEBUGGER` system handle. For more information, see Debugging from an ABL procedure on page 45.

---

To debug an application:

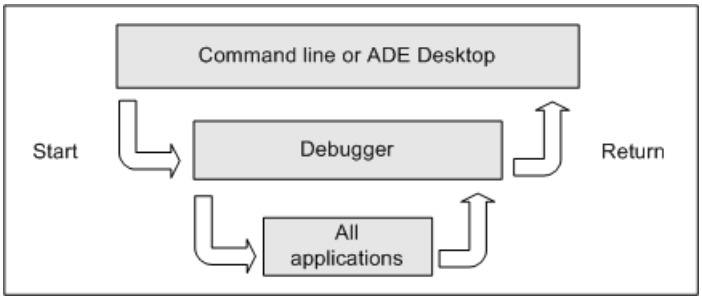
1. Load the procedure into the Debugger window, using **File** > **Open**.

   **Note:** If you start your OpenEdge session with the Startup Procedure (`-p`) startup parameter, the ABL Virtual Machine (AVM) loads the procedure into the Debugger window automatically.

2. Start the procedure, using **Debug** > **Run**.

   The following figure shows the relationship between the Debugger in stand-alone mode, the application procedures it runs, and the OpenEdge environment in which it is running.

   **Figure 4: Debugging in stand-alone mode**




When you exit any of these applications, it returns control to the Debugger. When you exit the Debugger, it completely removes itself and any application it is running in the OpenEdge environment, and returns control to the environment where you started it. If you restart the Debugger, for example, from the ADE Desktop, it completely re-initializes itself without any application in the Debugger context.

## Attachable mode

When you first start the Debugger and then attach it to a running client process, the Debugger runs in attachable mode. The client can be any ABL client, WebSpeed agent, or AppServer running on the same machine or on the same LAN as the Debugger. The Debugger gets control immediately upon attaching to the process. Once attached, the Debugger works the same as in application mode; that is, you are debugging an already running application.

By running the Debugger in attachable mode you can debug:

- A previously started session
- A session on another machine
- The AppServer, WebSpeed, asynchronous AppServer procedures, OpenClient, and Web Services directly from the Debugger without adding statements to the source code to start the Debugger

To successfully attach the Debugger, the process you attach to must be in a debug-ready state. For more information, see Making the process ready for debugging on page 36.

### Requirements for r-code

To support debugging in attachable mode, it is important to compile the r-code according to these rules:

- Use the `DEBUG-LIST` option with the `COMPILE` statement, and specify a location to which the Debugger has access as the location for the debug listing.
- Do not use `MIN-SIZE` option with the `COMPILE` statement. This option strips the Debugger Segment from the generated r-code.

For more information about these options, see the section on the `COMPILE` statement in *ABL Reference*.

### Performance and security considerations

Running an attachable mode debugging session has both security and performance considerations. The Debugger and the AVM instance communicate by opening a socket on a port and establishing a connection. In attachable mode, the AVM has to open a socket on a known port and wait for a connection from the Debugger. The open connection introduces a potential security gap and a performance degradation. To close the security gap, OpenEdge requires that you have physical or login access to the local machine to make a process debug-ready. To avoid the performance hit of unnecessary polling, you can make the process debug-ready by opening the connection only when you are ready to debug the process.

## Making the process ready for debugging

Before you can start the Debugger in attachable mode, you have to ready the process for debugging. The process can be any ABL (GUI or character mode) client, WebSpeed agent, or AppServer process, but not a WebClient™ process. You can make the process ready for debugging by:

- Using the Debugger window on page 36
- Using the Ready Attachable Debugger (-debugReady) startup parameter on page 36
- Using the proDebugConfig command on page 37

While a process is debug-ready, it might run slower than normal. Therefore, you should turn on this capability only when you intend to use the Debugger in attachable mode.

### Using the Debugger window

If you are running the Debugger on the same machine as the process to be debugged, you can make a process debug-ready through the Debugger window. In this case, the user interface mirrors what you do when you run the `proDebugConfig` command. You provide the same information and receive the same message feedback. For more information, see Attach to Process dialog box on page 63.

### Using the Ready Attachable Debugger (-debugReady) startup parameter

The Ready Attachable Debugger (`-debugReady`) startup parameter makes a process ready for debugging at startup. For example:

```
-debugReady port-number
```

*port-number*

A port number or zero (0). If you specify a known port number, the AVM attempts to use that port to create a socket to await a connection from the Debugger. If the port is in use, the AVM reports the error and continues to run, without making the process ready for debugging.

If you specify 0, the AVM determines a usable port number and reports that number to you during startup processing, as follows:

- For a GUI client, in an Alert box (even if running in batch mode)
- For a WebSpeed agent, in the WebSpeed log file
- For an AppServer process, in the AppServer log file
- For a client process running on UNIX, in the shell window (even if running in batch mode)

If you attempt to ready debugging on startup and debugging is not enabled for the installation (see Enabling debugging on page 25), you receive a warning message, and then startup continues. If you specify `-debugReady` and `-debug` in the same startup parameter, the AVM reports this as invalid and shuts down.

---

**Note:** All WebSpeed agents and AppServer processes started by the same broker use the same startup parameter. If you use such a broker configuration, do not specify a port number with the `-debugReady` startup parameter.

---

## Using the proDebugConfig command

The `proDebugConfig` command gives you a way to instruct a running ABL client, WebSpeed agent or single AppServer process to make itself ready for debugging. You must execute the command on the same machine as the process you are enabling. You can run the command from either a UNIX shell or a Windows **Proenv** shell (**Programs** > **Proenv** from the **Start** menu). The command has the following syntax:

## Syntax

```
proDebugConfig [process Id] [-ready {yes | no}] [-port port-number]
```

*process Id*

The ID of the process you want to enable for debugging. If you don't specify an ID, the command searches the local machine and displays a list of running OpenEdge processes and prompts you to enter an ID shown in the list.

On UNIX, the list might resemble the following:

```
mscat 16273     1  0   Jul 08 pts/37  0:03 /pro/dlc/bin/_progres
   -p /usr1/InvApp/main.p -T /tmp -h 10
mrdog 9295  26391  0 10:57:38 pts/40  0:01 /pro/dlc/bin/prowin32
   -pf startup.pf
```

In Windows, the list might resemble:

```
C:\pro\dlc\bin\_progres.exe   16273
C:\pro\dlc\bin\prowin32.exe    9295
```

Enter the process ID, for example, **9295**, and press **ENTER**. Press **CTRL+C** to cancel the command.

`-ready { yes | no }`

Turns the debug-ready state on or off. If you specify `yes`, and the process is currently ready for debugging (either from a previous command execution or the `-debugReady` startup parameter), the command displays a message to that effect that includes the port number used for the connection. If you specify `no`, and the ready state was already turned off by a previous command execution, or was never ready to begin with, the command displays a message to that effect. The default is `yes`.

`-ready { yes | no }`

Turns the debug-ready state on or off. If you specify `yes`, and the process is currently ready for debugging (either from a previous command execution or the `-debugReady` startup parameter), the command displays a message to that effect that includes the port number used for the connection. If you specify `no`, and the ready state was already turned off by a previous command execution, or was never ready to begin with, the command displays a message to that effect. The default is `yes`.

*`port-number`*

A port number or zero (0). If you specify a known port number, the AVM attempts to use that port to create a socket to await a connection from the Debugger. If the port is in use, `proDebugConfig` reports the error and the AVM instance continues to run, without making the process ready for debugging. If you do not specify the parameter or you specify 0, the AVM determines a usable port number, which the command displays to you in the shell.

If the command succeeds in making the process debug-ready, it displays a message to that effect and terminates. If the command execution is unsuccessful, the command outputs a message to the shell, the contents of which might indicate one of the following:

- You entered an incorrect process ID.
- The process you are trying to ready for debugging might be from an earlier release or a non-OpenEdge process.
- The AVM instance is busy, perhaps performing an extremely slow query, waiting on a database lock or a response from an AppServer, and is thus unable to respond in a timely manner. The attempt, in this case, times out after 25 seconds.
- The AVM instance is unable to respond, pending completion of one of the following statements:
  - `MESSAGE VIEW-AS ALERT BOX` (GUI Windows)
  - `READKEY` (character mode Windows)
  - `PAUSE` (character mode Windows)

  The command displays a message, instructing you to satisfy the condition and retry the operation.
- You tried to change the debug-ready state of a process that is already in a debug session.

For a WebSpeed agent, when the command successfully alters the debug-ready state, it writes the status (on or off) to the WebSpeed log file.

For an AppServer process, when the command successfully alters the debug-ready state, it writes the status (on or off) to the server log file.

If the AppServer is currently waiting for a new request, the Debugger will get control on the next line of executable ABL code.

For more information on starting the debugger in attachable mode, see Detaching from the process on page 46.

## Stacked applications

In all three modes, you can stack applications. That is, you can run a new application at any interruption point of an executing application. When you do so, the previous application waits on the procedure call stack. The new application can access any shared or global data defined by previous applications stacked in the Debugger context. When the new application exits, it returns control to the Debugger, and the previous application is ready to continue at its current interruption point. When the Debugger exits, it closes all applications stacked in the Debugger context before returning to its startup environment.

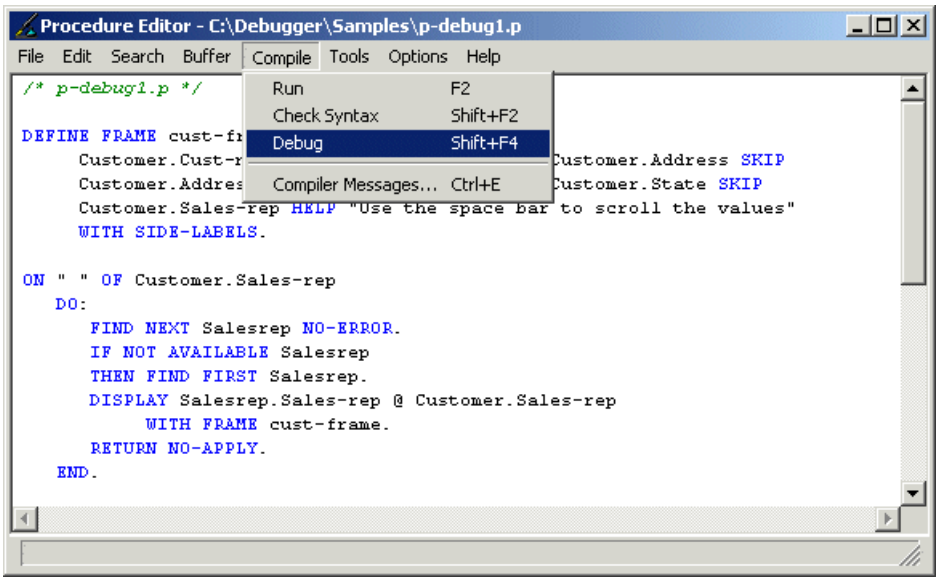
## Debugging from the Procedure Editor

Starting the Debugger from the Procedure Editor runs the Debugger in application mode, ready to debug the procedure that is running and any other procedures you choose.

To start a debugging session from the Procedure Editor:

Enter or open a procedure in the Editor buffer and choose **Compile** > **Debug**. The following figure shows the Procedure Editor in Windows.

**Figure 5: Debugging from the Procedure Editor**




After you choose **Compile** > **Debug**, the Procedure Editor:

1. Saves a temporary file containing your procedure to the current working directory, using a unique, alphanumeric filename with a `.ped` extension (as the Procedure Editor always does)
2. Initializes the Debugger

3. Sets a breakpoint on the first executable line of your procedure
4. Runs your procedure

The AVM instance immediately gives control to the Debugger at the initial breakpoint.

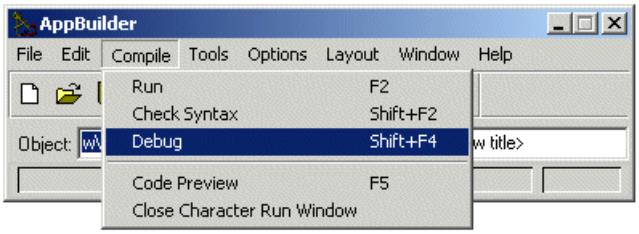
# Debugging from the AppBuilder

Starting the Debugger from the AppBuilder runs the Debugger in application mode, ready to debug the procedure that is running and any other procedures you choose.

To start a debugging session from the AppBuilder:

Open an application procedure and choose **Debug** > **Compile**, as shown in the following figure.

**Figure 6: Debugging from the AppBuilder in Windows**




After you choose **Compile** > **Debug**, the AppBuilder:

1. Saves a temporary file containing your procedure to the current working directory, using a unique, alphanumeric filename with a `.ab` extension (as the AppBuilder always does)
2. Initializes the Debugger
3. Sets a breakpoint on the first executable line of your procedure
4. Runs your procedure

The AVM instance immediately gives control to the Debugger at the initial breakpoint.

# Debugging from the ADE Desktop

Starting the Debugger from the ADE Desktop runs the Debugger in stand-alone mode, ready to execute and debug any procedures you open in the Debugger window.

---

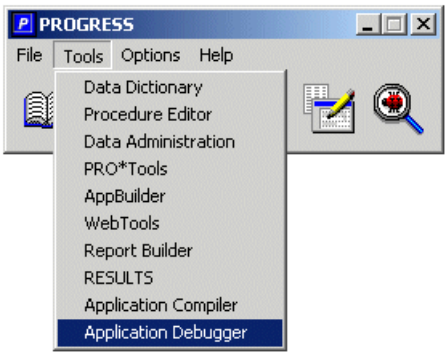
**Note:** The Debugger does not automatically set a breakpoint on the first executable line of procedures you execute from the Debugger when you start it from the ADE Desktop. Be sure to set at least one breakpoint before running a procedure.

---

To start a debugging session from the ADE Desktop:

choose **Tools** > **Application Debugger** or click on the **Application Debugger** button on the toolbar. The following figure shows the **Application Debugger** menu option and toolbar button in Windows.

**Figure 7: Debugging from the ADE Desktop**




The ADE Desktop immediately initializes and gives control to the Debugger, without an initial application running. Starting from the ADE Desktop runs the Debugger in stand-alone mode, ready to execute and debug any procedures you open in the Debugger window. Note the Debugger does not automatically set a breakpoint on the first executable line of procedures you execute from the Debugger when you start it from the ADE Desktop. Be sure to set at least one breakpoint before running a procedure.

---

**Note:** You can use the **Interrupt** option from the Debugger window to stop execution of most any procedure, whether or not it has a breakpoint set. For more information on the **Interrupt** and **Breakpoints** options, see Debugger Window and Files on page 47

---

# Debugging from the OpenEdge command line

Starting the Debugger from the command line using the Debugger startup parameter (`-debug`) runs the Debugger in stand-alone mode.

- To start a debugging session from the OpenEdge command line:

  Run the client session with the Debugger (`-debug`) startup parameter, as in the following example:

  ```
  pro sports -debug
  ```

  The Debugger is the entry point for all application procedures you want to run. The Debugger starts and allows you to open a procedure and set breakpoints in the procedure before running it, using **Debug** > **Run**.

- To start a single-user debugging session from the OpenEdge command line and start debugging the `arrmenu.p` procedure:

  ```
  pro sports -p arrmenu.p -debug
  ```

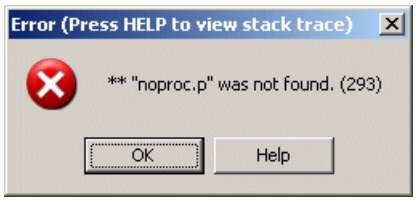
  This command runs the Debugger in stand-alone mode, ready to execute and debug `arrmenu.p` (or any other procedures you choose). An automatic breakpoint is set on the first executable line of `arrmenu.p`.

# Debugging from alert dialog boxes

When an unhandled ABL error or .NET exception occurs—there is no `CATCH` or `NO-ERROR` logic present—in an interactive session for any ABL client, the ABL Virtual Machine (AVM) displays an Error message box. When Debug Alert is set to `TRUE`, the message box also includes a **Help** button. The **Help** button is also present on ABL Alert box messages. Clicking on the **Help** button produces a Stack Trace dialog box containing ABL stack trace and .NET stack trace information. There is also a **Debug** button on this dialog box for initiating the Debugger.

To enable the Debug alert feature, use the Debug Alert (`-debugalert`) client startup parameter or set `SESSION:DEBUG-ALERT` to `TRUE`. The following figure shows an alert dialog box with a **Help** button.

**Figure 8: Alert dialog box with Stack Trace Help button**




**Note:** For more information on the Debug Alert startup parameter, see the reference entry in *Startup Command and Parameter Reference*. For more information on session handles, see Using System Handles for Debugging on page 89

When you click the **Help** button, a **Stack Trace** dialog box opens, displaying the ABL stack trace for the client session currently executing. This information is helpful to determine where unhandled errors are occurring in your code. In addition to the normal ABL stack trace, the CallStack displays the .NET stack trace if the error occurs due to .NET. You can also get the CallStack information from a thrown ABL error object if it initiates the unhandled error. The most recent routine call is displayed at the top of each respective trace listing for the ABL stack, the .NET stack trace, and the error CallStack.

The following procedure describes the steps to display the stack trace feature.

## Displaying the .NET Stack Trace in the Stack Trace dialog box

To display the .NET Stack Trace in the Stack Trace dialog box:

1. Start a Progress Developer Studio for OpenEdge session. If you want to see the .NET stack trace, the session must be running on Windows.
2. Run the ABL code specified below in Progress Developer Studio for OpenEdge. For more information, see *Progress Developer Studio for OpenEdge Online Help*.

```
USING System.Windows.Forms.*.

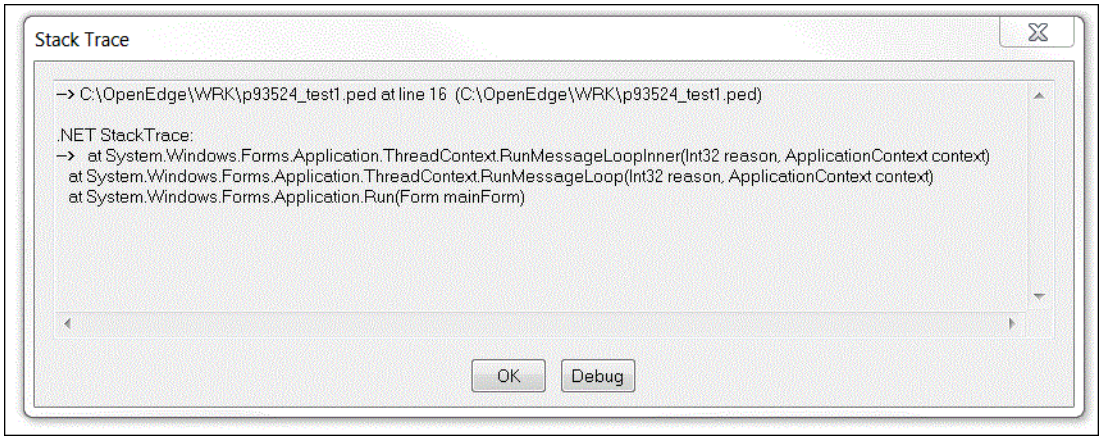
DEFINE VARIABLE mainForm AS Form   NO-UNDO.
DEFINE VARIABLE form2    AS Form   NO-UNDO.
DEFINE VARIABLE dlgForm  AS Form   NO-UNDO.
DEFINE VARIABLE btn      AS Button NO-UNDO.
ASSIGN
  mainForm      = NEW Form()
  mainForm:Text = "Main and Only Form"
  btn           = NEW Button().

mainForm:Controls:ADD(btn).
btn:Click:Subscribe("BtnClick").
WAIT-FOR Application:Run(mainForm).
MESSAGE "Done" VIEW-AS ALERT-BOX INFO BUTTONS OK.
PROCEDURE btnClick:
  DEFINE INPUT PARAMETER sender AS Button         NO-UNDO.
  DEFINE INPUT PARAMETER args   AS MouseEventArgs NO-UNDO.
  RUN newForm.
END.
PROCEDURE newForm:
  form2 = NEW Form().
  WAIT-FOR Application:Run(form2). /* This should fail w/.NET Exception */
END.
```

3. **Compile** the code and run it. The Main and Only Form window appears.
4. Click on the button. A .NET exception Error message box appears.
5. Click **Help** in the Error message box. The **Stack Trace** dialog box appears containing the .NET stack trace.

The following figure shows a .NET **Stack Trace** dialog box with the.NET stack trace.

**Figure 9: Stack Trace Dialog box including .NET Stack**




## Displaying the ABL Error CallStack in the Stack Trace dialog box

To display the ABL Error CallStack in the **Stack Trace** dialog box:

1. Start a Progress Developer Studio for OpenEdge session.
2. Run the ABL code specified below in Progress Developer Studio for OpenEdge. For more information, see *Progress Developer Studio for OpenEdge Online Help*.

```
Main.p
SESSION:DEBUG-ALERT=YES.
SESSION:ERROR-STACK-TRACE=YES.
Run New.p
/*Error is raised here and displayed on the screen.*/

New.p
ROUTINE-LEVEL ON ERROR UNDO, THROW.

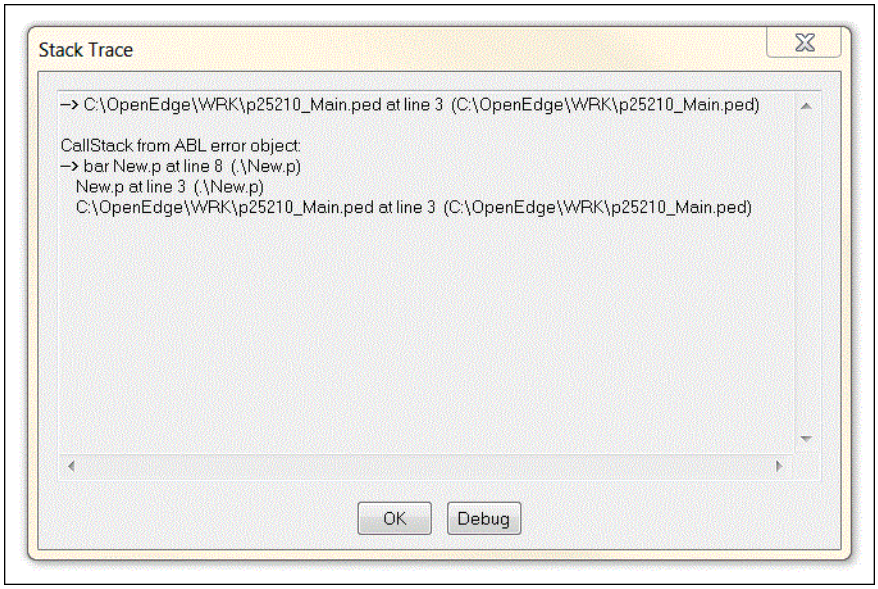
RUN BAR.
/*Error is automatically thrown here due to the UNDO, THROW directive or it
could be caught with a CATCH block and re-thrown with the same effect. */

PROCEDURE bar:
FIND CUSTOMER WHERE Name BEGINS "Z".
/*This fails */
/*Error is automatically thrown here due to the UNDO, THROW directive or it
could be caught with a CATCH block and re-thrown with the same effect. */
END.
```

3. Compile the code and run it.
4. Click Help in the Error message box. The **Stack Trace** dialog box appears. It contains the ABL Stack Trace and displays the CallStack information under the heading "CallStack from ABL error object".

The following figure shows a ABL **Stack Trace** dialog box with the ABL stack trace.

**Figure 10: Stack Trace dialog box including the ABL CallStack**



Use the -clientlog startup parameter to enable the display of CallStack information in a log file. Each line of the trace gets prefixed by the standard log file line header. The following figure displays the appearance of CallStack information in a log file.

**Figure 11: CallStack information in a log file**

```
[11/04/20@16:22:31.258-0400] P-010908 T-006844 1 4GL -- ** ABL Debug-Alert Stack Trace **
[11/04/20@16:22:31.258-0400] P-010908 T-006844 1 4GL -- --> Main.p at line 3  (Main.p)
[11/04/20@16:22:31.258-0400] P-010908 T-006844 1 4GL --
[11/04/20@16:22:31.258-0400] P-010908 T-006844 1 4GL -- CallStack from ABL error object:
[11/04/20@16:22:31.258-0400] P-010908 T-006844 1 4GL -- --> bar New.p at line 8  (New.p)
[11/04/20@16:22:31.258-0400] P-010908 T-006844 1 4GL --     New.p at line 3  (New.p)
[11/04/20@16:22:31.258-0400] P-010908 T-006844 1 4GL --     Main.p at line 3  (Main.p)
```

For more information on the format of each stack line, see *ABL Reference.*

## Starting the Debugger from the Stack Trace dialog box

To start the Debugger from the Stack Trace dialog box:

1. In the **Alert** dialog box, click **Help** to open the **Stack Trace** dialog box.

   **Note:** You must enable debugging in Prowin32 on Windows or from a UNIX shell before you click Debug in the Stack Trace dialog box. If debugging is not enabled, the system raises an error and it does not allow you to debug the code.

2. Click **Debug** in the **Stack Trace** dialog box. The **Stack Trace** dialog box is closed.
3. The original error **Alert** dialog box must be dismissed by clicking OK before the Debugger gets control. The Debugger will show the current statement as the line that will execute next after the error is handled.

**Note:** You cannot use the Debugger if the original error stops the application.

# Debugging from an ABL procedure

Starting the Debugger from an ABL procedure using the DEBUGGER or SESSION system handle runs the Debugger in either application or stand-alone mode.

The DEBUGGER system handle provides attributes and methods that allow you to start and control a debugging session from an ABL procedure. It is intended for experienced ABL programmers who want to:

Control breakpoints at specific points in their code, regardless of how the surrounding code is modified, or set breakpoints in include files that might appear in many different procedures.

- Develop their own ABL or OpenEdge development tools, and initialize and control Debugger interaction from those tools.
- Debug AppServer and WebSpeed applications.

The SESSION system handle allows you to specify whether you want alert dialog boxes to include a **Help** button that provides access to a **Stack Trace** dialog box and the Debugger. For more information on starting the Debugger from alert dialog boxes, see Debugging from alert dialog boxes on page 42.

For information on using the DEBUGGER and SESSION system handles, see Using System Handles for Debugging on page 89.

# Debugging in attachable mode

Starting the Debugger, then attaching to a debug-ready client process, runs the Debugger in attachable mode. Be sure to read Attachable mode on page 35.

To start the Debugger in attachable mode, follow the instructions below, based on your platform:

- **Windows** — From the Start menu, choose **Programs** > **OpenEdge** > **Debugger**.
- **UNIX** — Change directory (`cd`) to `$DLC/bin`. At the prompt, type `proDebugger`.

When you start the Debugger in attachable mode, the user interface appears the same as in other modes. However, the only options available are:

- To make changes to preferences use the **Preferences** dialog box. For more information on the **Preferences** dialog box, see Preferences dialog box on page 83.
- To attach the Debugger to a process use the **Attach to Process** dialog box. For more information on the **Attach to Process** dialog box, see Attach to Process dialog box on page 63.

For more information on making a process ready for debugging, see Making the process ready for debugging on page 36.

## Detaching from the process

After successfully attaching to a running client process, the **Debug** > **Attach to Process** menu item becomes **Debug** > **Detach from Process**. When you choose this menu option, the Debugger disconnects from the socket it uses to communicate with the AVM instance. If the debug-ready state was initiated by the Debugger, it then asks if you want to leave the process in a debug-ready state. Choose **Yes** or **No**, as appropriate. **No** is the default.

After you detach from the process, the Debugger returns to its startup state, with the menu item restored to **Debug** > **Attach to Process**. You can now attach to another process or close the Debugger.

If you attempt to close the Debugger window while it is attached to a process, the Debugger asks for confirmation. If you confirm, the Debugger disconnects from the process and might ask if you want to leave the process in a debug-ready state. Answer **Yes** or **No**. The Debugger then shuts down.

---

**Note:** In attachable mode, the Debugger remains open if you close the OpenEdge session, or when the current application completes, unlike other operating modes, where the Debugger itself shuts down as well.

---

# 3

# Debugger Window and Files

The Debugger window is your primary means of interacting with the Debugger. It operates independently of any window used by your ABL (Advanced Business Language) application. From the Debugger window, you can control most Debugger features, from setting breakpoints and watches to listing and executing procedures under Debugger control.

The Debugger window provides panes, menus, toolbar buttons, and dialog boxes that you use to invoke Debugger commands and view results. You can configure various window settings and save them for future debugging sessions.

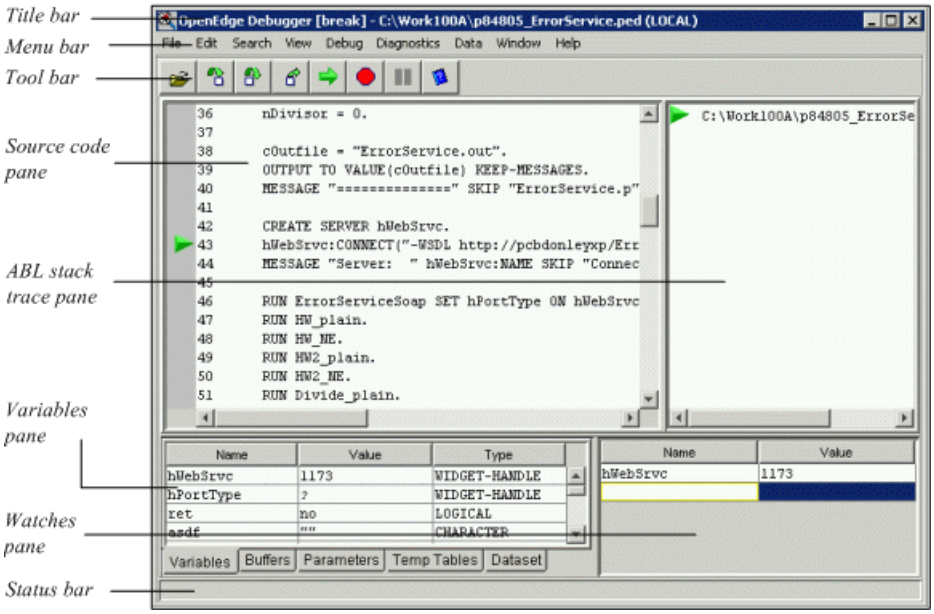
For details, see the following topics:


- Debugger window layout
- Debugger window panes
- Debugger title bar
- Debugger menu bar options
- Debugger toolbar buttons
- Debugger status bar
- Debugger dialog boxes
- Debugger files

# Debugger window layout

The following figure shows the layout of the Debugger window.

**Figure 12: Debugger window**




The layout of the Debugger window generally is the same on all supported systems; any significant differences are described in this chapter. The exact dimensions and layout of your Debugger window might differ, depending on the size and resolution of your monitor. However, the general layout of the window is fixed.

The Debugger window contains four panes, as well as a title bar, menu bar, toolbar, and status bar. You can resize the Debugger window like any other window on your system. You can also resize each pane vertically or horizontally in relation to the pane next to it. To resize the panes in Windows, point to the resize bar between the panes. When the pointer changes to a double-headed arrow, drag the resize bar up, down, right, or left. To resize the panes on UNIX, select the resize handles between the panes and drag them up, down, right, or left. Each pane provides scroll bars, as required, to allow you to access portions of the pane that might be vertically or horizontally out of view.

All panes except the source code pane can be hidden. To show or hide a Debugger window pane, select or clear the corresponding menu option on the **View** menu. For more information, see View menu on page 57.

The Debugger automatically saves these window settings and lets you set other Debugger window settings in the **Preferences** dialog box. For more information, see Preferences dialog box on page 83.

The Debugger updates all panes when it takes control of an executing procedure. For example, the stack trace pane does not reflect the current ABL stack trace while an application is executing; the Debugger updates the stack trace pane only when it encounters a breakpoint or when you step through code.

# Debugger window panes

The following sections describe the Debugger window panes and how to use them.

## Source code pane

The *source code pane* displays a read-only debug listing including breakpoints and the current line of execution. It can contain the executing procedure or any other procedure whose listing you want to see. The name of the procedure whose listing the Debugger displays appears in the Debugger title bar. For more information, see Debugger title bar on page 56.

The source code pane has a context (right-click) menu containing the following options: **Copy**, **Find**, **Find Next**, **Find Previous**, **Add Breakpoint**, **Remove Breakpoint**, **Enable Breakpoint**, **Disable Breakpoint**, **Run to Line**, and **Add Watch** (adds the selected item to the watches pane).

### Debug listing

The *debug listing* is a specially prepared source code listing that lets you set breakpoints and follow the course of procedure execution in an application. It is an expanded listing file that the ABL compiler creates during procedure compilation.

The format of a debug listing is similar to the standard ABL compiler listing. Like compiler listings, debug listings expand all preprocessor, include file, and parameter references. The main difference is that the text in debug listings contains only numbered source code lines without headings, page formatting, or any of the other internal data information (such as buffer and frame use) usually provided by compiler listings.

Line numbers appear in the source code pane if you select the **Include Source Line Numbers** option in the **Preferences** dialog box. For more information, see Preferences dialog box on page 83. The Debugger displays line numbers by default.

The Debugger displays the listing for the executing procedure each time it receives control (for example, at a breakpoint). If the Debugger cannot find the debug listing for the procedure, it tries to generate it from the procedure source code. If the Debugger can find only the r-code for the procedure, it displays the procedure name and line number in the status bar as you step through the procedure.

For more information on procedure source files and debug listing files derived from procedure files, see Debugger files on page 86.

### Listing indicators

When the source code pane displays the listing for the executing procedure or any procedure on the call stack, it also displays two listing indicators: a *line pointer* that shows the next line to execute and a *breakpoint indicator* that shows each line where a breakpoint is set. The following table lists the Debugger listing indicators.

**Table 2: Listing indicators**

| Icon | Description |
|---|---|
|  | The line pointer indicator for the executing procedure appears in green to the left of the next executable line. When you are viewing a procedure on the call stack other than the active procedure, the line pointer appears in yellow. |
|  | The breakpoint indicator appears in red to the left of each line where a breakpoint is set. Disabled breakpoints appear as red outlines. For information on maintaining breakpoints in a procedure, see Breakpoints dialog box on page 64. |

### Text selection

You can select one full or partial line of source code to set watches or copy to the clipboard.

### Run to Line command

The **Run to Line** command continues program execution until it reaches a selected line of ABL code. Execution resumes with the selected line. If the Debugger encounters a breakpoint or interrupt before reaching the selected line, execution stops, and the Debugger cancels the command. The execution pointer is at the break or interrupt line.

If the selected line is not an executable line of code, the Debugger stops execution on the first executable line after the selected line. If there are no executable lines of code on or after the selected line, the Debugger stops execution on the last executable line of code in the procedure file.

If program execution ends before the Debugger reaches the selected line, the Debugger cancels **Run to Line**, and the debugging session ends.

To issue **Run to Line**, right-click on the chosen line in the source code pane and choose **Run to Line** from the context (right-click) menu.

You can select a line in any procedure file open in the Debugger. The selected line does not have to be in the current procedure.

## ABL stack trace pane

The *ABL stack trace pane* displays the ABL stack trace for the executing procedure. This pane indicates what procedure you are in and what procedure you came from. The top of the pane shows the top of the stack. You can investigate the code associated with a line of the stack trace by clicking on the stack trace line. The Debugger automatically updates the source code pane, variables pane, and watches pane to reflect the code and data associated with the selected stack trace line.

## Variables pane

The *variables pane* provides access to the variables associated with the current procedure. The Debugger organizes and displays variable data on tabs named for the following categories:

- Variables
- Buffers
- Parameters
- Temp Tables
- Dataset

Choose a tab to display all variables in that category. Each tab in the variables pane has a context (right-click) menu containing the following options: **Copy**, **Cut**, **Paste**, **Dataview** (displays attribute or field information for the selected variable in the **Dataview** dialog box), and **Add Watch** (adds the selected variable to the watches pane). For more information, see Dataview dialog box on page 65 and Debugger dialog boxes on page 62.

Double-clicking on a cell containing the value of a variable makes the value editable. You can modify the value by entering a different value and pressing the **ENTER** key or clicking out of the cell. Pressing the **ESC** key when editing a value restores the current value and cancels the editing operation.

The Debugger displays raw (unformatted) variable data. For example, a decimal value of 1,234.56 appears as 1234.56; 1.00 appears as 1. Do not use formatting characters when entering variable values. Dynamic temp-table and dataset-related objects appear in the variables tab as handle variables.

## Accessing array variables and parameters

Array variables and parameters appear on their respective tabs in the variables pane. The Debugger displays the name of the array, the number of elements in the array (in brackets), and the data type of the array (for example, `DaysInMonth [12] INTEGER`). You can examine the value of a single array element as a watch (`DaysInMonth[3]`), or examine the value of all elements in an array in the **Array** tab of the **Dataview** dialog box.

If an array has an indeterminate extent (that is, it was declared as `EXTENT` with no limiting expression), the Debugger displays the current value of the `EXTENT` function for the array. Before the array's size is determined, the extent of the array is unknown, so the Debugger displays `[?]` in the **Value** column for the variable or parameter. When the array is populated or received as a parameter from an array with a known size, the extent is set.

For example, if a calling procedure declares the array as `EXTENT 5` and passes it to a procedure where the array was declared to have an indeterminate extent, the receiving array is set to an extent of 5, once the procedure is called. At this point, the Debugger displays `[5]` as the value, and there is no indication the array originally was declared as having an indeterminate extent.

## Accessing BLOB fields

You use `BLOB` (Binary Large OBject) fields to store large amounts of arbitrary binary data. A `BLOB` is similar to a `MEMPTR`, and the Debugger handles `BLOB`s in much the same way as `MEMPTR`s. Specifically, the Debugger displays the total length of a `BLOB` (in parentheses) followed by the first 256 bytes of data in the `BLOB` (as a hexadecimal string). If a `BLOB` contains more than 256 bytes of data, the Debugger displays an ellipsis indicating there is more data than shown. For example:

```
(133496) 64 42 74 65 0A 69 20 6D 20 64 75 65 2C 65 21 0D ...
```

You cannot edit `BLOB` values; the **Value** cell is read-only.

Due to its small size, the **Value** cell shows only a small part of the `BLOB` data. To see the full 256 bytes of `BLOB` data, examine the `BLOB` in the **Value** tab of the **Dataview** dialog box.

## Accessing CLOB fields and LONGCHAR variables

You use `CLOB` (Character Large OBject) fields and `LONGCHAR` variables to store large amounts of character data. The Debugger displays the total length of a `CLOB` or `LONGCHAR` (in parentheses) followed by the first 256 characters in the `CLOB` or `LONGCHAR` (as a quoted string). If a `CLOB` or `LONGCHAR` contains more than 256 characters, the Debugger displays an ellipsis indicating there is more data than shown.

You cannot edit `CLOB` and `LONGCHAR` values; the **Value** cell is read-only.

Due to its small size, the **Value** cell shows only a small part of `CLOB` and `LONGCHAR` data. To see the full 256 characters of data, examine the `CLOB` or `LONGCHAR` in the **Value** tab of the **Dataview** dialog box.

## Accessing DATETIME and DATETIME-TZ variables

You use a `DATETIME` variable to store an ABL date and time, and a `DATETIME-TZ` variable to store an ABL date and time, as well as a time zone value. The Debugger displays `DATETIME` and `DATETIME-TZ` variables based on the current date format specified by the `DATE-FORMAT` attribute for the session (typically `mdy` or `dmy`). The Debugger also interprets `DATETIME` and `DATETIME-TZ` variable assignments in this way.

You can assign a new value to a `DATETIME` or `DATETIME-TZ` variable by editing its value.

## Accessing class object instances

You can access instances of user-defined classes, as well as the following types of elements that make up a class:

- Data members, which define instance data of a class
- Properties, which are similar to data members, and also provide `GET` and/or `SET` accessors containing ABL code that executes implicitly when a property is read or written
- Methods, which define the behavior of a class

The data members, properties, and methods that a class comprises are statically defined in a class and scoped to an instance of that class, and they are available inside or outside of the class hierarchy depending on their access mode (PUBLIC, PRIVATE, or PROTECTED). When accessing data members and properties, or stepping through methods, the Debugger honors their access mode.

---

**Note:** `PRIVATE` members only appear when the class is the current execution frame on the ABL stack. `PUBLIC` and `PROTECTED` members appear from any execution frame that references them.

---

For more information about user-defined classes, and their data members, properties, and methods, see *Develop Object-oriented ABL Applications*.

The Debugger displays the object reference for an instance of a user-defined class as an `INTEGER` value followed by the name of the instance (in parentheses). For example "`1101 (Class MyClass)`".

You can examine and edit the value of data members and properties for an instance of a user-defined class in the variables pane, the watches pane, or the **Dataview** dialog box.

## Accessing object properties

A property typically defines and implements a `GET` and/or `SET` accessor, which provides a class object instance with read and write access to the property's data value, respectively. However, a property can define a `GET` and/or `SET` accessor without an implementation. In this case, the accessor reads the value from or writes the value to the property's default memory (that is, a default storage area that holds the property's current value). At the very least, a property must define either a `GET` accessor or a `SET` accessor in order to read or write the property's value.

If the property defines and implements a `GET` or `SET` accessor, the accessor implementation executes implicitly. When you step into code that reads or writes a property's value, the Debugger steps into the ABL code of that accessor implementation.

In certain cases, the code structure is such that errors occur when the `GET` accessor implementation executes. This happens, for example, if the implementation uses a buffer handle, and the Debugger executes the accessor before the handle is assigned. To prevent this problem, you can disable the evaluation by the Debugger of properties that have non-empty accessor methods. To do so, start the OpenEdge Client session with the `-noevalprops` startup parameter. For more information, see *Startup Command and Parameter Reference*.

Consider the following Debugger behavior when examining and editing property values:

- If the property only defines a `GET` or `SET` accessor, and does not provide an implementation for the accessor, the Debugger reads the value from or writes the value to the property's default memory.
- If the property does not define a `GET` accessor, the Debugger cannot read the property's value and displays "`** Unavailable **`" in the property's **Value** cell.
- If the property does not define a `SET` accessor, the Debugger cannot write the property's value (the **Value** cell is read-only).

- If the property defines only a `SET` accessor, and no `GET` accessor, the Debugger can write the property's value but will immediately display "`** Unavailable **`" in the property's **Value** cell.
- If you started the OpenEdge session with the `-noevalprops` startup parameter, the **Value** cell displays "`** Property evaluation disabled **`" for those properties that have a `GET` method with an implementation. In this case, you can inspect the value of a property in the **Dataview** dialog box (see Dataview dialog box on page 65.) Note that ABL properties with empty `GET` methods are always evaluated.

---

**Note:** If a property defines and implements a `GET` or `SET` accessor, the value of the property can change each time the Debugger reads or writes the property because the accessor implementation will execute each time, and its logic might change the property's value. For example, a property's `GET` accessor might calculate the property's value based on the current time.

---

### Accessing ProDataSet objects

The ProDataSet object is a collection of temp-table buffer objects and relations among the temp-table objects. A ProDataSet consists of three object types—`DATASET`, `DATA-RELATION`, and `DATA-SOURCE`—each with several attributes and methods. The **Dataset** tab contains the names of all static ProDataSet-related objects declared in the current procedure context. You can right-click on any listed object and choose **Dataview** from the pop-up menu, to see the object's attributes.

## Watches pane

The *watches pane* keeps track of variables, buffer fields, attributes, data, and built-in ABL functions on which a watch is set. For more information on functions as watches, see Watches for built-in functions on page 55.

The watches pane has a context (right-click) menu containing the following options: **Copy**, **Cut**, **Paste**, **Add Watch** (adds a new variable to the watches pane), **Remove Watch**, and **Dataview** (displays attribute or field information for the selected variable in the **Dataview** dialog box).

You can add a watch by doing any of the following:

- Entering the watch into the last line of the watches pane directly.
- Entering the watch in the **Add Watch** dialog box. For more information, see Debugger dialog boxes on page 62.
- Selecting text in the source code pane, or selecting a variable or parameter in the variables pane, and then choosing the **Add Watch** option on the context (right-click) menu.
- Selecting text in the source code pane and dragging it to the watches pane. For more information, see Watches pane on page 53.
- Selecting an entity in the **Dataview** dialog box and choosing the **Add Watch** button.

Note the following with respect to validity checking of the selected text:

- Selected text that is improperly formatted displays "`** Error **`" in its **Value** cell.
- Selected text that is properly formatted but that refers to an out-of-scope or nonexistent variable displays "`** Unavailable **`" in its **Value** cell.

Valid watch formats are any of the following:

| |
|---|
| [ VARIABLE \| FIELD \| PARAMETER ] *name* |
| *handle*:*attribute* |
| *object-reference* [ [*array-index*] ] [ :*data-member* [ [*array-index*] ] \| *property* ] |
| [ *data-member* [ [*array-index*] ] \| *property* ] |
| *widget-name* [ [*array-index*] ] :*attribute* [ IN FRAME *frame-name* ] |
| { FRAME \| MENU } *widget-name*:*attribute* |
| { SUB-MENU \| MENU-ITEM } *widget-name*:*attribute*<br>[ IN { MENU \| SUB-MENU } *menu-name* ] |

You can chain attribute and property references. For example, the following attribute reference displays the value of the `TITLE` attribute for a window containing a fill-in (assuming the fill-in is three levels from the window):

```
FILL-IN-1:PARENT:PARENT:PARENT:TITLE
```

If a variable is currently out of scope, or you add a watch for an unknown variable name, the following message appears in the **Value** cell:

```
** Unavailable **
```

If evaluation of properties is disabled (by means of the `-noevalprops` startup parameter), the following message appears in the **Value** cell for properties that have a non-empty accessor method (ABL properties that have an empty `GET` method are always evaluated):

```
** Property evaluation disabled**
```

**Note:** You can use the **Value** tab of the Dataview dialog box to inspect the value of a property, whether or not evaluation of properties is disabled. See Dataview dialog box on page 65 for more information.

Double-clicking on a cell containing the value of a variable makes the value editable. You can modify the value by entering a different value and pressing the **ENTER** key or clicking out of the cell. Pressing the **ESC** key when editing a value restores the current value and cancels the editing operation.

You can remove a watch for a variable by selecting the watch in the watches pane and then:

- Pressing the **DELETE** key
- Choosing the **Remove Watch** option on the context (right-click) menu or **Debug** menu

## Watches for built-in functions

You can add a watch to a built-in ABL function to track the status of the function.

To add a new watch, type the complete function name directly in the watches pane or choose **Debug** > **Add Watch**. The Debugger displays the function and its value in the Watches pane.

The following table lists the ABL functions supported by the Debugger.

**Table 3: ABL functions supported by the Debugger**

| Functions that can accept assignment of a new value | Functions that cannot accept assignment of a new value |
|---|---|
| CURRENT-LANGUAGE<br>PROMSGS<br>PROPATH | _PCONTROL<br>_SERIAL-NUM<br>COM SELF<br>DATASERVERS<br>DBNAME<br>ETIME<br>FRAME-DB<br>FRAME-FIELD<br>FRAME-INDEX<br>FRAME-NAME<br>FRAME-VALUE<br>GO-PENDING<br>LASTKEY<br>MACHINE-CLASS<br>MESSAGE-LINES<br>NUM-ALIASES<br>NUM-DBS<br>OPSYS PROCESS-ARCHITECTURE<br>OS-DRIVES<br>OS-ERROR<br>PROGRESS<br>PROVERSION<br>RETRY<br>RETURN-VALUE<br>SCREEN-LINES<br>TERMINAL<br>TIME<br>TODAY<br>TRANSACTION<br><br>**Note:** TERMINAL is settable only on UNIX character interfaces. The values of TIME and TODAY change independently of program execution, so the values displayed in the watches pane might not agree with the last values displayed in the ABL program. |

You can modify a function's value by editing the **Value** cell in the Watches pane. If the function does not accept an assignment, the function retains its value. In this case, the Debugger does not generate an error message.

**Note:** You cannot watch a function that requires arguments.

## Drag-and-drop watches

You can add a watch by dragging an entity to the watches pane.

To add a watch by dragging and dropping:

1. Select the text that you want to add a watch for in the source code pane (by clicking and dragging with the mouse to select the text).
2. Drag the selected text to the watches pane.
3. Release the mouse button to add the object as a watch to the last (empty) row in the watches pane. If you release the mouse over any other pane or outside the Debugger window, the watch is not added.

# Debugger title bar

The Debugger title bar displays the name of the procedure whose listing the Debugger is displaying. The title bar also displays the status of the procedure in brackets, and the location of the procedure in parentheses. When the procedure is executing, the status is [run]. When the Debugger has control, the status is [break]. When the procedure is local, the location is (`LOCAL`). When the procedure is remote, the location is (`REMOTE`). If the status and location do not appear in the title bar, the procedure has not yet started executing.

Any procedure you run in application mode from a Procedure Editor buffer is displayed using a filename with a `.ped` extension. This is the filename for a temporary procedure file where the Procedure Editor writes the ABL contents of the buffer. The names of temporary procedure files written by the AppBuilder have a `.ab` extension.

# Debugger menu bar options

Like other Application Development Environment (ADE) tools, the primary way to access Debugger features and functions is through menu options on pull-down menus on the menu bar. You can access menu options in more than one way, using the mouse to choose menu options, using accelerator keys and mnemonics to invoke menu options directly from the keyboard, or using tool bar buttons for more frequently used menu options.

The following sections describe the Debugger menus and how to access their options.

## Accelerator keys and mnemonics

Some menu options have predefined *accelerator keys* labeled to the right of each menu option. Accelerator keys are function keys you can press to choose a menu option.

Most menu options have a *mnemonic*, a single letter you can press to choose a menu option after you choose the menu. The mnemonic for each menu option is underlined in its label.

## File menu

The **File** menu provides the standard means to open a file or end your debugging session. The following table describes the **File** menu options.

**Table 4: File menu**

| Command | Purpose |
|---|---|
| Open... (**F3**) | Displays the **Open** dialog box, which lets you open a procedure file in the source code pane. For more information, see Open dialog box on page 83. |
| Exit | Terminates the Debugger session and returns control to the environment or application in which the Debugger was invoked. |

## Edit menu

The **Edit** menu lets you cut, copy, and paste a specified text selection, as well as set breakpoints and preferences. The following table describes the **Edit** menu options.

**Table 5: Edit menu**

| Command | Purpose |
|---|---|
| Cut (**CTRL+X**) | Cuts the text currently selected in the variables or watches pane and places it on the clipboard. |
| Copy (**CTRL+C**) | Copies the currently selected text in the source code, variables, or watches pane to the clipboard. |
| Paste (**CTRL+V**) | Pastes the contents of the clipboard to the specified position in the variables or watches pane. |
| Breakpoints... (**CTRL+B**) | Displays the **Breakpoints** dialog box, which lets you maintain breakpoints in a procedure. For more information, see Breakpoints dialog box on page 64. |
| Preferences... | Displays the **Preferences** dialog box, which lets you set user preferences for the Debugger window. For more information, see Preferences dialog box on page 83. |

## Search menu

The **Search** menu lets you find a specified text string in your debug listing. The following table describes the **Search** menu options.

**Table 6: Search menu**

| Command | Purpose |
|---|---|
| Find... (**CTRL+F**) | Displays the **Find** dialog box, which lets you specify text you want to find in the source code pane. For more information, see Find dialog box on page 77. |
| Find Next (**F9**) | Finds the next occurrence of the specified text. |
| Find Prev (**SHIFT+F9**) | Finds the previous occurrence of the specified text. |

## View menu

The **View** menu lets you show and hide various Debugger window panes. Select or clear an option to show or hide the corresponding pane, respectively. The following table describes the **View** menu options.

**Table 7: View menu**

| Command | Purpose |
|---|---|
| Variables | Shows or hides the variables pane |

| Command | Purpose |
|---|---|
| Stack Trace | Shows or hides the ABL stack trace pane |
| Watches | Shows or hides the watches pane |

## Debug menu

The **Debug** menu controls procedure execution and Debugger-application interaction. The following table describes the **Debug** menu options.

**Table 8: Debug menu**

| Command | Purpose |
|---|---|
| Run (**F2**) | Starts or restarts execution of the procedure currently displayed in the source code pane (when invoked by choosing **File > Open** or using the Debugger (`-debug`) and Startup Procedure (`-p`) startup parameters). |
| Continue (**F5**) | Continues execution of the procedure until it encounters a breakpoint, an interrupt, or the end of the procedure. |
| Step Into (**F7**) | Executes the next statement in a procedure. If the statement calls a subprocedure or causes a trigger to execute, the Debugger steps into the subprocedure or trigger and stops at the first executable line of code. The trigger or subprocedure is now the current procedure, and you can continue execution from the current stopping point.<br><br>**Note:** The Debugger does not step into the destructor method of an object that has been automatically deleted by garbage collection, unless the destructor contains a breakpoint. In cases where the object is explicitly deleted by means of the `DELETE OBJECT` statement, the Debugger does step into the destructor.and Step Into command |
| Step Over (**F6**) | Executes the next statement in a procedure. If the statement calls a subprocedure, or causes a trigger to execute, the Debugger executes the trigger or subprocedure without stepping into it and stops at the next executable line of code. If the Debugger encounters a breakpoint in a subprocedure or trigger, execution stops at the breakpoint, the executing subprocedure or trigger becomes the current procedure, and you can continue execution from the current breakpoint.<br><br>**Note:** After stepping over a statement, if the next statement is in a destructor method for an object that has been deleted, the Debugger does not stop in the destructor unless the destructor contains a breakpoint. This is true whether the object is deleted automatically by garbage collection or explicitly by means of the `DELETE OBJECT` statement. |

| Command | Purpose |
|---|---|
| Step Out (**SHIFT+F7**) | Continues execution until the current procedure ends, returning control to the calling procedure. The execution pointer is at the next line of the calling procedure. If the Debugger encounters a breakpoint or interrupt before returning to the calling procedure, the Debugger cancels the command. The execution pointer is at the break or interrupt line. If issued from the top-level procedure on the ABL call stack and the Debugger encounters no breakpoints or interrupts before the procedure ends, the debugging session ends. The exception is if you start the Debugger with `-debug`, in which case the Debugger has control, waiting for you to launch a new ABL program. |
| Stop | Immediately terminates the executing application as if it invoked the ABL `STOP` statement at the current break or interrupt point, and rolls back uncommitted changes in the current transaction. |
| Interrupt (**CTRL+CANCEL**) | Interrupts and halts an executing procedure and gives control to the Debugger without setting a breakpoint. |
| Attach To Process... | Opens the **Attach to Process** dialog box, where you can specify a running client process to optionally make debug-ready and attach to for the purpose of debugging. This option is only available when you open the Debugger in attachable mode. For more information, see Attach to Process dialog box on page 63. Changes to **Detach From Process**, after the Debugger attaches to a process. |
| Start Animation | Enables visual feedback as the Debugger automatically steps through code line by line, by updating the watches and variables panes with current values.When specified with **Step Into**, the Debugger steps into `RUN` statements; with **Step Over**, the Debugger steps over `RUN` statements. After each line executes, the Debugger pauses briefly. Use **Edit** > **Preferences** to increase or decrease the pause. |
| Stop Animation | Terminates animated stepping. Unavailable until animated stepping is in progress. |
| Dataview... (**CTRL+D**) | Displays the **Dataview** dialog box, which lets you examine attribute and field data for a variable. For more information, see Dataview dialog box on page 65. |
| Add Watch... | Displays the **Add Watch** dialog box, which lets you add a watch for a variable in a procedure. For more information, see Watches pane on page 53. |
| Remove Watch | Removes the selected watch. |
| Show Transaction | Displays a message indicating whether there is a transaction active at the current line of the executing procedure. |

## Diagnostics menu

The **Diagnostics** menu lets you access Debugger diagnostics tools. The following table lists the diagnostics tool available on the **Diagnostics** menu.

**Table 9: Diagnostics menu**

| Command | Purpose |
|---|---|
| Monitor Dynamic Objects... | Displays the **Dynamic Object Tracking** dialog box, which lets you monitor the creation and destruction of dynamic object instances in an ABL application. For more information, see Dynamic Object Tracking dialog box on page 70. |

## Data menu

The **Data** menu lets you select an ABL system handle and display the attributes associated with that handle in the **Dataview** dialog box. You can modify the values of writable attributes.

---

**Note:** Some handles might be unknown at various stages of execution. For example, the `FOCUS` and `SELF` handles typically are valid only when a trigger is executing.

---

For more information on the **Dataview** dialog box, see Dataview dialog box on page 65. For more information on ABL system handles and their associated attributes, see *ABL Reference*.

## Window menu

The **Window** menu lists the open files, and lets you switch to another open file. Switching to another open file does not affect the state of the executing procedure.

## Help menu

The **Help** menu provides access to help information about the Debugger window components and your Debugger installation. The following table describes the **Help** menu options.

**Table 10: Help menu**

| Command | Purpose |
|---|---|
| Help Topics (**F1**) | Displays information about Debugger window components |
| About | Displays version information about the Debugger |

# Debugger toolbar buttons

The Debugger toolbar contains shortcut buttons for frequently used menu options. The following table describes the toolbar buttons.

**Table 11: Toolbar buttons**

| Button | Action |
|---|---|
|  (Open) | Displays the **Open** dialog box, which lets you open a procedure file. For more information, see Open dialog box on page 83. |

| Button | Action |
|---|---|
|  (Step Into) | Executes the next statement in a procedure. If the statement calls a subprocedure or causes a trigger to execute, the Debugger steps into the subprocedure or trigger and stops at the first executable line of code. The trigger or subprocedure is now the current procedure, and you can continue execution from the current stopping point.<br><br>**Note:** The Debugger does not step into the destructor method of an object that has been automatically deleted by garbage collection, unless the destructor contains a breakpoint. In cases where the object is explicitly deleted by means of the `DELETE OBJECT` statement, the Debugger does step into the destructor.and Step Into command |
|  (Step Over) | Executes the next statement in a procedure. If the statement calls a subprocedure or causes a trigger to execute, the Debugger executes the trigger or subprocedure without stepping into it and stops at the next executable line of code. If the Debugger encounters a breakpoint in a subprocedure or trigger, execution stops at the breakpoint, the executing subprocedure or trigger becomes the current procedure, and you can continue execution from the current breakpoint.<br><br>**Note:** After stepping over a statement, if the next statement is in a destructor method for an object that has been deleted, the Debugger does not stop in the destructor unless the destructor contains a breakpoint. This is true whether the object is deleted automatically by garbage collection or explicitly by means of the `DELETE OBJECT` statement. |
| (Step Out) | Continues execution until the current procedure ends, returning control to the calling procedure. The execution pointer is at the next line of the calling procedure. If the Debugger encounters a breakpoint or interrupt before returning to the calling procedure, the Debugger cancels the command. The execution pointer is at the break or interrupt line. If issued from the top-level procedure on the ABL call stack and the Debugger encounters no breakpoints or interrupts before the procedure ends, the debugging session ends. The exception is if you start the Debugger with `-debug`, in which case the Debugger has control, waiting for you to launch a new ABL program. |
| (Continue) | Continues execution of the specified procedure until it encounters a breakpoint, an interrupt, or the end of the procedure. |
| (Stop) | Immediately terminates the executing application as if it invoked the ABL `STOP` statement at the current break or interrupt point, and rolls back uncommitted changes in the current transaction. |
| (Stop Animation) | Turns off animated stepping. Appears only when animated stepping is in progress. |
| (Interrupt) | Interrupts and halts an executing procedure and gives control to the Debugger without setting a breakpoint. |
| (Help Topics) | Displays information about Debugger window components. |

# Debugger status bar

The Debugger status bar appears at the bottom of the Debugger window and displays the following types of information:

- Debugger status messages
- Procedure name and line number

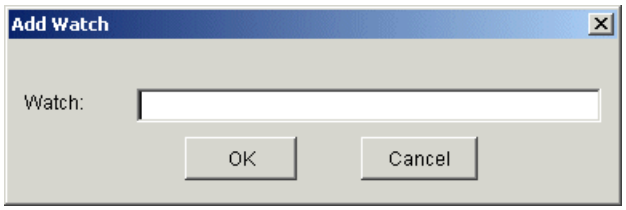
# Debugger dialog boxes

The following sections provide usage information for each of the Debugger dialog boxes and are presented in alphabetical order.

## Add Watch dialog box

The **Add Watch** dialog box lets you add a watch for any variable, including fields and handle or object attributes, in the executing procedure or any procedure on the call stack. A watch allows you to keep track of the value of a variable, parameter, buffer field, attribute, data, or built-in ABL function as you navigate through your ABL application. The name and value of the variable on which the watch is set appears in the watches pane. The following figure shows the **Add Watch** dialog box.

**Figure 13: Add Watch dialog box**




Access this dialog box by choosing the **Add Watch** option on the **Debug** menu or the context (right-click) menu in the watches pane.

To add a watch:

Enter an entity in the **Watch** field and click **OK**. See Watches pane on page 53 for a list of supported watch expressions.

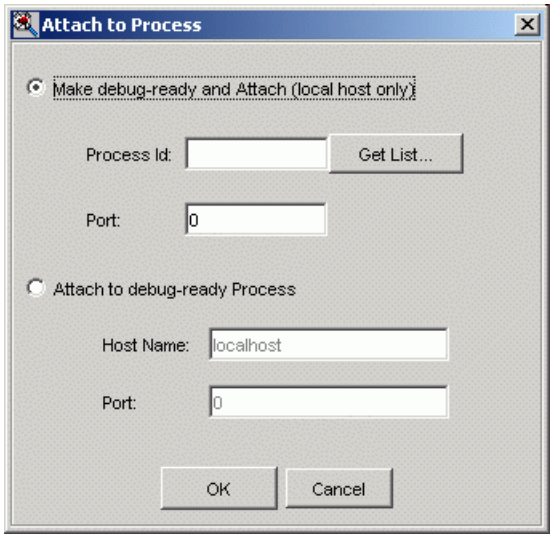
You can also add a watch for an entity by:

- Entering the watch in the watches pane directly
- Selecting text in the source code pane or selecting a variable in the variables pane, and then choosing the **Add Watch** option on the context (right-click) menu
- Selecting an entity in the **Dataview** dialog box and choosing the **Add Watch** button
- Dragging selected text from the source code pane and dropping it onto the watches pane

## Attach to Process dialog box

The **Attach to Process** dialog box lets you attach the Debugger to a running client process. Optionally, you can make the process debug-ready and attach to it in a single step.For more information on making a process debug-ready, see Making the process ready for debugging on page 36. The following figure shows the **Attach to Process** dialog box.

**Figure 14: Attach to Process dialog box**




Access this dialog box by choosing **Debug** > **Attach** > **to Process** after starting the Debugger in attachable mode. Select either of the following options:

- **Make debug-ready and attach (local host only)** — Makes a process debug-ready and attaches the Debugger to the process. Select this option only if the process is running on the same machine as the Debugger. Choosing the **Get List** button lets you select an existing process in the **Select Process** dialog box. For more information, see Select Process dialog box on page 86.
- **Attach to debug-ready process** — Attaches the Debugger to a process you have already made debug-ready using either the `-debugReady` startup parameter or the `proDebugConfig` command. Select this option if the ABL Virtual Machine (AVM) instance is not running on the same machine as the Debugger.

### Using the Make debug-ready and Attach option

To make a process debug-ready and attach to it:

1. Select the **Make debug-ready and Attach (local host only)** radio button.
2. Enter the **Process ID** of the process on the local machine you want to debug.

   To list the currently running OpenEdge processes, click the **Get List** button. The **Select Process** dialog box appears. For more information, see Select Process dialog box on page 86.

3. Enter a port number that you want to use for communicating with the client process, or leave the default (0), for the AVM to determine a usable port number.
4. Click **OK**.

If the connection is unsuccessful, the Debugger displays an alert box reporting the reason for the failure. For information on possible reasons for the failure, see the *Using the proDebugConfig command* section in Attach to Process dialog box on page 63.

Upon a successful connection, the Debugger displays a message showing the port number that it is using for the connection, in case you need to reconnect, for any reason. After the dialog box closes, the AVM instance gives control to the Debugger, and the listing for the currently running procedure appears in the source code pane, at the point where the Debugger connection was acknowledged. For hidden procedures, the source code pane appears empty, but the Debugger announces the connection and indicates that the AVM instance will break at the first line of application code. Debugging can proceed as described for all other modes.

For an AppServer process or WebSpeed agent, if the Debugger successfully alters the debug-ready state, it writes the status (on or off) to the log file.

### Using the Attach to debug-ready Process option

To attach to a process that is already in a debug-ready state:

1. Select the **Attach to debug-ready Process** radio button.
2. Enter the name of the host on which the process is running, or accept the default **localhost**. You must enter a host name if the process is not running on the same machine as the Debugger.
3. Enter the port number used to make the process debug-ready. For more information, see Making the process ready for debugging on page 36.
4. Click **OK**.

After successfully connecting, the dialog box closes, and the AVM instance gives control to the Debugger. The listing for the currently running procedure appears in the source code pane, at the point where the Debugger connection was acknowledged. For hidden procedures, the source code pane appears empty, but the Debugger announces the connection and indicates that the AVM instance will break at the first line of application code. Debugging can proceed as described for all other modes.

If the attempt fails because the process was not in a debug-ready state, the Debugger reports only that the connection attempt timed out (it does not recognize that the process was not in a debug-ready state). The attempt can also fail because you specified an incorrect host name or port.

---

**Note:** In attachable mode, when the top-level procedure terminates, possibly returning control to the Procedure Editor or AppBuilder, the Debugger window does not automatically hide, but remains in view.

---

If control does return to the application development environment, you restart debugging on the application by choosing **Compile** > **Debug**. At that point the AVM instance gives control to the Debugger, stopping at the first executable line of the application, just as it would if you had not attached.

## Breakpoints dialog box

The **Breakpoints** dialog box lets you add, edit, remove, enable, and disable breakpoints or watchpoints. This dialog box displays all breakpoints set in the current debugging session.

A breakpoint interrupts program execution. There are three types of breakpoints:

- **At Line** — A breakpoint set at a specific line in a named procedure
- **On Error** — A breakpoint set to interrupt execution if an error occurs
- **Watchpoint** — A breakpoint set to interrupt program execution when the value of a variable, buffer field, or attribute reference changes

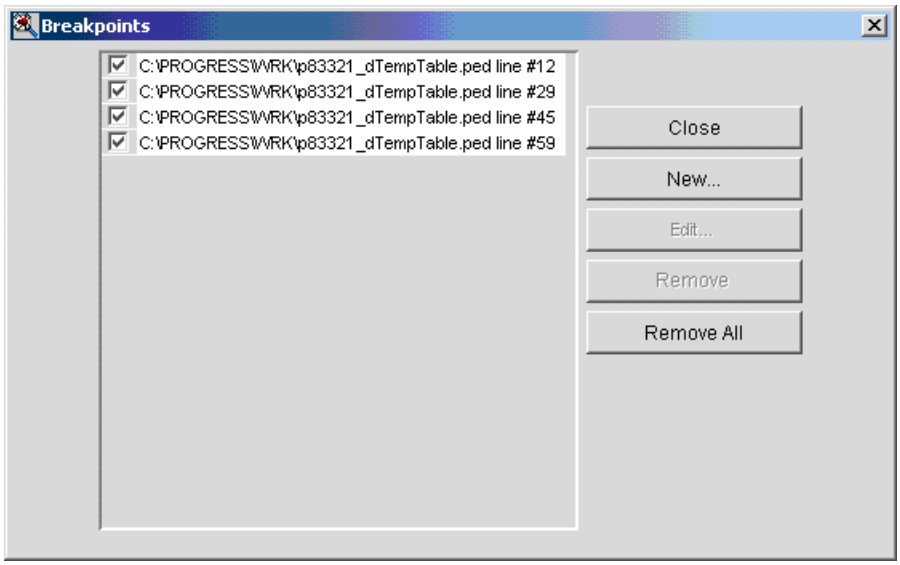
---

**Note:** A *watchpoint* stops program execution when an the value of a variable, buffer field, or attribute reference changes. A *watch* tracks values of variables, buffer fields, attributes, data and built-in ABL functions.

---

The following figure shows the **Breakpoints** dialog box.

**Figure 15: Breakpoints dialog box**




Access this dialog box by choosing **Edit** > **Breakpoints** or entering **CTRL+B**.

In this dialog box, you can perform the following actions:

- To enable or disable a breakpoint or watchpoint, select or clear the check box to the left of the breakpoint and click the **Close** button. Disabled breakpoints appear in the source code pane as red outlines.
- To add a new breakpoint or watchpoint, click the **New** button. The **New Breakpoints** dialog box appears. You can also add a breakpoint directly in the source code pane by clicking in the shaded area to the left of a code line. For more information on adding new breakpoints, see New Breakpoint dialog box on page 77.
- To edit a breakpoint or watchpoint, select the breakpoint or watchpoint you want to edit and click the **Edit** button. The **Edit Breakpoint** dialog box appears. For more information on editing breakpoints, see Edit Breakpoint dialog box on page 76.
- To remove a single breakpoint or watchpoint, select it and click the **Remove** button. You can also remove a single breakpoint by clicking on its breakpoint indicator in the source code pane.
- To remove all breakpoints and watchpoints, click the **Remove All** button.

## Dataview dialog box

The **Dataview** dialog box lets you examine and modify data for variables and arrays, attributes of objects, and field data for buffer objects. You can add a watch for any entity that appears in the dialog box.

Access this dialog box by choosing one of the following:

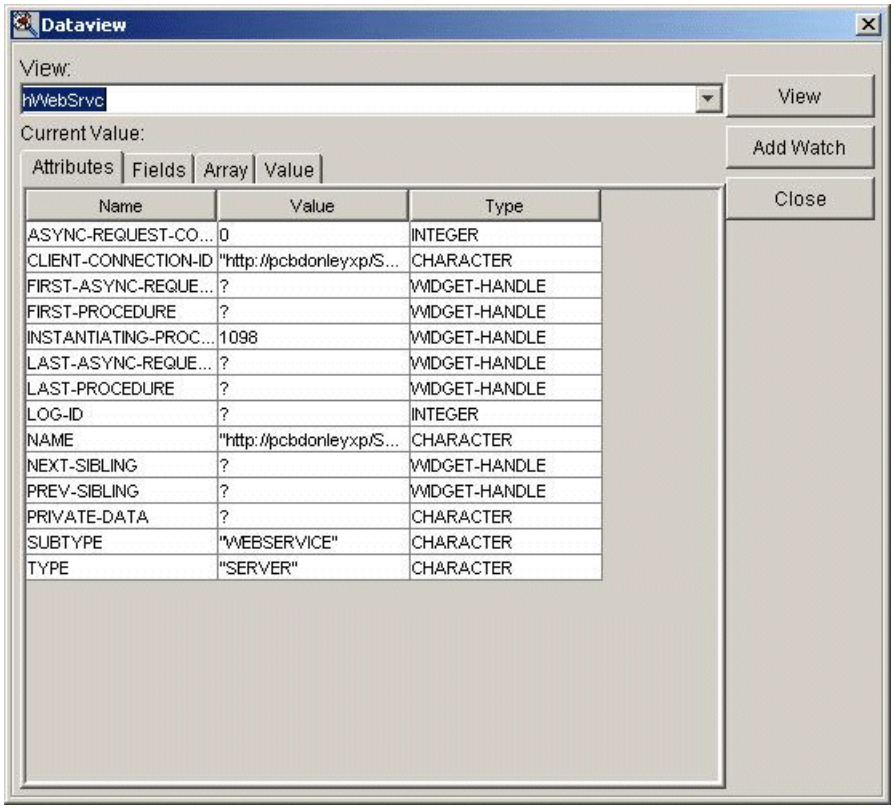
- The **Dataview** option on the **Debug** menu, or press **CTRL+D**
- The **Dataview** option on the context (right-click) menu from the watches or variables pane
- An ABL system handle on the **Data** menu

If you select an object before choosing the **Dataview** option, the object appears automatically in the **Dataview** dialog box. Otherwise, you can enter an object name in the **View** field or select an object you viewed previously from the drop-down list, and click the **View** button.

## Attributes tab

The **Attributes** tab lists the attributes associated with the selected object. The following figure shows the **Attributes** tab.

**Figure 16: Dataview dialog box with Attributes tab**

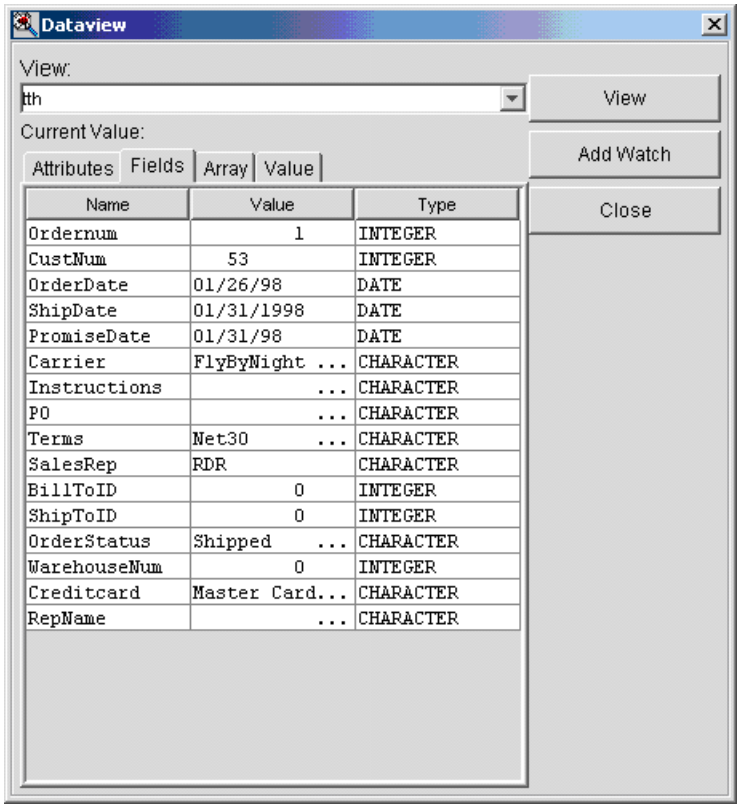



For write-only attributes, the **Value** column displays `** Unavailable **`.

## Fields tab

The **Fields** tab lists the fields associated with the selected object. The following figure shows the **Fields** tab.

**Figure 17: Dataview dialog box with Fields tab**




You can view buffer details, by selecting the buffer in the **Variable** pane, right-clicking to choose **Dataview**, and then choosing the **Field** tab to view all field values in the buffer. You can also add a watch.

Double-clicking on a cell in the value column makes the value editable. You can modify the value of variables, attributes (writable only) or fields by entering a different value and pressing the **ENTER** key. Pressing the **ESC** key when editing a value restores the current value and cancels the editing operation.

You can modify a field value in a buffer when the buffer contains a record; however, the modified value is not immediately stored in the database. To store the modified value in the database, you must execute the appropriate statement in your ABL code (such as `RELEASE` *`record`*).

Some attributes are handles to other objects (for example, the `PARENT` attribute of an object is the handle to the parent of that object). In this case, you can "walk the object tree" by selecting the attribute and choosing the **View** button. The new attribute appears in the **View** field (for example, if you are viewing `FILL-IN-1` and select the `PARENT` attribute, the new attribute appears as `FILL-IN-1:PARENT`). If the new attribute is a valid object handle, choosing the **View** button refreshes the data view for that attribute. You can access the original object by selecting it from the drop-down list and choosing the **View** button.

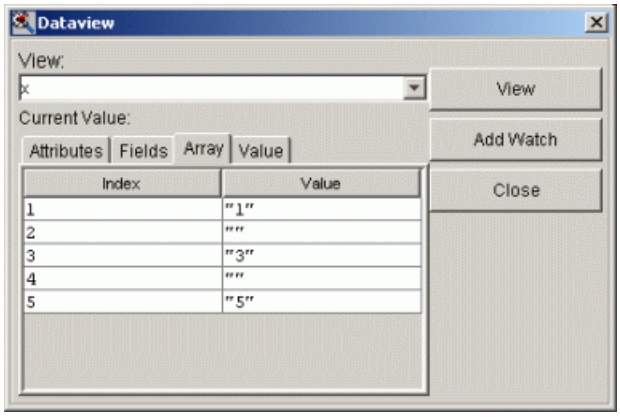
You can chain object references. For example, the following object reference displays the information for a window containing a fill-in (assuming the fill-in is three levels from the window):

```
FILL-IN-1:PARENT:PARENT:PARENT
```

## Array tab

The **Array** tab displays the elements of an array (a variable, parameter, or field defined with the `EXTENT` option) and their values. To view the elements of an array, right-click on an array in the variables pane and choose **Dataview** from the pop-up menu. The **Dataview** dialog box appears with the **Array** tab populated with the elements of the selected array, as shown in the following figure.

**Figure 18: Array elements in the Dataview Array tab**




You can also view the elements of an array field in a buffer. When viewing the buffer's fields in the **Dataview** dialog box, select the array field, click the **View** button, and then choose the **Array** tab to view the field's elements.

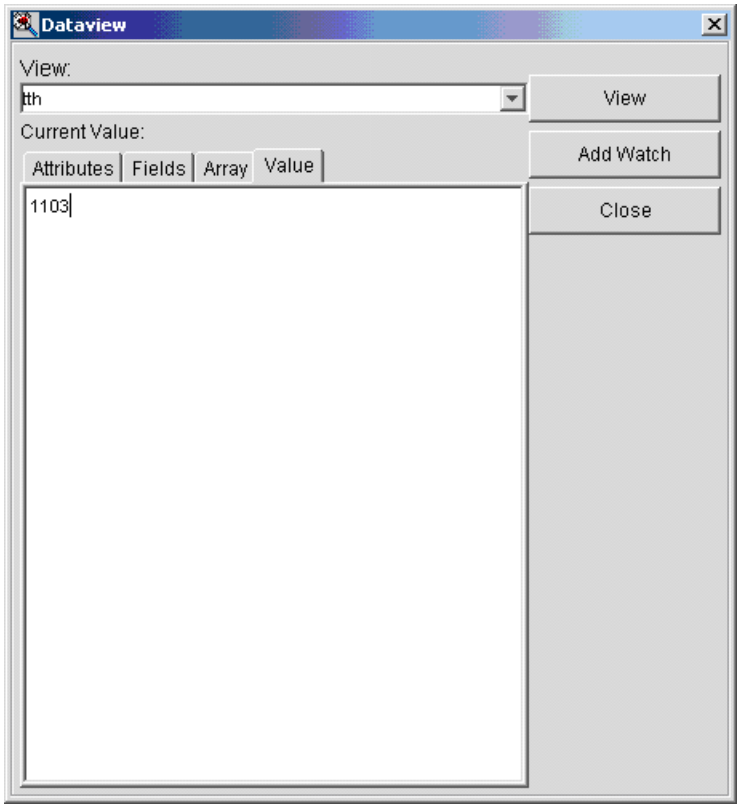
The array viewer can show a limited amount of data, approximately 30KB. If the **Array** tab cannot show every element of an array at once, it displays "**...**" as the value of the last element. Select this element and click the **View** button to refresh the **Array** tab. The array viewer then shows the next 30KB of the array, beginning at the selected element.

You can modify the elements of an array. Double-click on a cell containing the value to make it editable. Enter a different value and press the **ENTER** key. Pressing the **ESC** key when editing a value restores the current value and cancels the editing operation.

## Value tab

The **Value** tab enables you to view the value of a discrete entity without adding a watch. The following figure shows the **Value** tab.

**Figure 19: Dataview dialog box with Value tab**




**Note:** You access some system handle dataviews from the **Data** menu. These include the `LOG-MANAGER`, `SESSION`, and `ERROR-STATUS` system handles.

To view an entity:

1. Enter an entity name in the edit box. You can specify the following entities:
   - Variables
   - Attributes
   - Fields
   - Handles
   - Data members
   - Properties

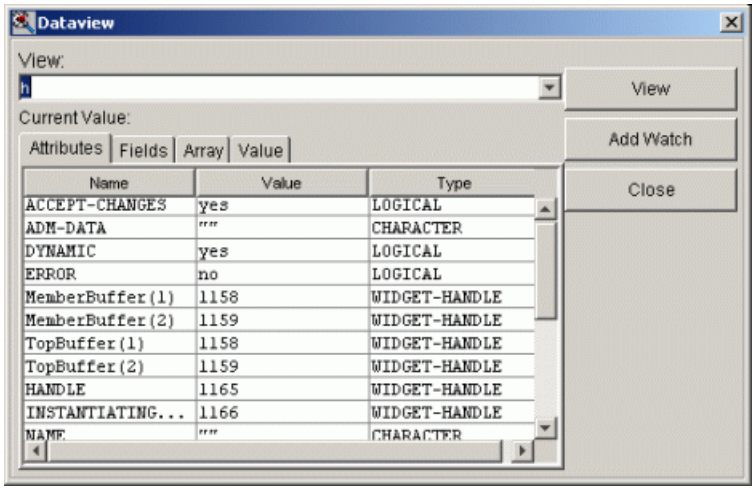
   You can enter expressions in colon syntax for attributes and properties (for example, `myWindow:TITLE` or `myVariable:property1`).
2. Click the **View** button. The entity's value appears in the **Value** tab.

You can modify the value of the entity. Double-click on the value to make it editable. Enter a different value and press the **ENTER** key. Pressing the **ESC** key when editing a value restores the current value and cancels the editing operation.

### Distinguishing ProDataSet methods from attributes

The Debugger displays references to the objects contained in a ProDataSet, in addition to the attributes of the ProDataSet itself. These objects are accessible via ProDataSet methods. The Debugger, however, cannot execute methods; therefore, in the **Dataview** dialog box, these references appear as special entries in mixed case (upper and lower), to differentiate them from the attributes, which appear in uppercase, as shown in the following figure.

**Figure 20: ProDataSet methods and attributes in the Dataview dialog box**




You can select one of the `MemberBuffer` lines and refresh the **Dataview** dialog box to view the buffer's attributes. The Debugger takes a similar approach to the other methods that retrieve ProDataSet-related information (`GET-CHILD-RELATION`, `GET-SOURCE-BUFFER`, and so on), as shown in the following table:

**Table 12: ProDataSet-related information**

| Object | Method | Dataview object name |
|---|---|---|
| `BUFFER` | `GET-CHILD-RELATION()` | `ChildDataRelation` |
| `DATA-SOURCE` | `GET-SOURCE-BUFFER()` | `SourceBuffer` |
| `DATASET` | `GET-BUFFER-HANDLE()` | `MemberBuffer` |
| `DATASET` | `GET-RELATION()` | `DataRelation` |
| `DATASET` | `GET-TOP-BUFFER()` | `TopBuffer` |

## Dynamic Object Tracking dialog box

The **Dynamic Object Tracking** dialog box lets you monitor the creation and destruction of dynamic object instances in an ABL application. Use this dialog box to locate dynamic object instances the application has created, no longer needs, and has not yet destroyed. These undeleted object instances might indicate a memory leak.

Access this dialog box by choosing **Diagnostics > Monitor Dynamic Objects**.

The **Dynamic Object Tracking** dialog box contains the following tabs:

- Control Monitoring tab on page 71 — Lets you select the types of dynamic objects you want to monitor
- View Objects tab on page 75 — Lists the active dynamic object instances, based on the selected object types

The **Dynamic Object Tracking** dialog box also contains the following buttons:

- **Start Monitoring button** — Starts the object monitoring process. When processing begins, the label on this button changes to **Restart Monitoring**.
- **Stop Monitoring button** — Stops the object monitoring process. When processing stops, the label on the **Restart Monitoring** button changes back to **Start Monitoring**.
- **Close button** — Closes the **Dynamic Object Tracking** dialog box.

## Control Monitoring tab

The **Control Monitoring** tab lets you select the types of dynamic objects you want to monitor. The following figure shows the **Control Monitoring** tab.

**Figure 21: Dynamic Object Tracking dialog box with Control Monitoring tab**



The **Control Monitoring** tab contains a treeview control that displays the types of dynamic objects you can monitor, organized within object categories. The following list shows the dynamic object types you can monitor, as they appear on the **Control Monitoring** tab:

**Table 13: Database Objects**

| | |
|---|---|
| QUERY (excluding query handles for static queries) | BUFFER (excluding buffer handles for static buffers) |
| TEMP-TABLE (excluding table handles for static temp-tables) | DATASET |
| DATA-SOURCE | – |

**Note:** `BUFFER-FIELD` objects are not listed, since an application has no control over their creation or destruction.

**Table 14: User Interface widgets**

| | |
|---|---|
| BROWSE | BROWSE-COLUMN (excluding handles associated with static browse objects) |
| BUTTON | COMBO-BOX |
| EDITOR | FILL-IN |
| FRAME | IMAGE |
| MENU | MENU-ITEM |
| RADIO-SET | RECTANGLE |
| SELECTION-LIST | SLIDER |
| SUB-MENU | TEXT |
| TOGGLE-BOX | WINDOW |

**Table 15: XML Objects**

| | |
|---|---|
| SAX-ATTRIBUTES | SAX-READER |
| SAX-WRITER | SOAP-HEADER |
| SOAP-HEADER-ENTRYREF | X-DOCUMENT |
| X-NODEREF | – |

**Note:** `SOAP-FAULT` and `SOAP-FAULT-DETAIL` objects are not listed, since an application has no control over their creation or destruction.

**Table 16: Other Objects**

| | |
|---|---|
| ASYNC-REQUEST | CALL |
| CONTROL-FRAME | MEMPTR (any memory allocated for a MEMPTR) |
| PROCEDURE (excluding handles, such as THIS-PROCEDURE, for non-persistent procedures) | SERVER |
| SERVER-SOCKET | SOCKET |

**Note:** For more information on these dynamic object types, see *ABL Reference*.

To expand a category node, click the plus sign (**+**) next to the node. To collapse a category node, click the minus sign (**–**) next to the node.

You can select all object types within a category by selecting the check box at the category node level, whether or not the node is expanded. You can also select one or more individual object types within a category, after you expand the category node, by:

- Selecting the category node check box, which selects all object types within that category, and clearing the check box for each object type you want to exclude
- Clearing the category node check box, which clears all object types within that category, and selecting the check box for each object type you want to include

To begin tracking the creation and destruction of the object instances for the object types you selected, click the **Start Monitoring** button. Then, use the **View Objects** tab to monitor the active object instances. An active object instance is an instance of a dynamic object the application has created, but not yet destroyed, since you started or restarted monitoring.

| | |
|---|---|
| QUERY (excluding query handles for static queries) | BUFFER (excluding buffer handles for static buffers) |
| TEMP-TABLE (excluding table handles for static temp-tables) | DATASET |
| DATA-SOURCE | – |

**Note:** `BUFFER-FIELD` objects are not listed, since an application has no control over their creation or destruction.

- **User Interface widgets**

| | |
|---|---|
| BROWSE | BROWSE-COLUMN (excluding handles associated with static browse objects) |
| BUTTON | COMBO-BOX |
| EDITOR | FILL-IN |
| FRAME | IMAGE |
| MENU | MENU-ITEM |
| RADIO-SET | RECTANGLE |

| | |
|---|---|
| SELECTION-LIST | SLIDER |
| SUB-MENU | TEXT |
| TOGGLE-BOX | WINDOW |

**Table 17: XML Objects**

| | |
|---|---|
| SAX-ATTRIBUTES | SAX-READER |
| SAX-WRITER | SOAP-HEADER |
| SOAP-HEADER-ENTRYREF | X-DOCUMENT |
| X-NODEREF | – |

**Note:** `SOAP-FAULT` and `SOAP-FAULT-DETAIL` objects are not listed, since an application has no control over their creation or destruction.

**Table 18: Other Objects**

| | |
|---|---|
| ASYNC-REQUEST | CALL |
| CONTROL-FRAME | MEMPTR (any memory allocated for a MEMPTR) |
| PROCEDURE (excluding handles, such as THIS-PROCEDURE, for non-persistent procedures) | SERVER |
| SERVER-SOCKET | SOCKET |

**Note:** For more information on these dynamic object types, see *ABL Reference*.

To expand a category node, click the plus sign (**+**) next to the node. To collapse a category node, click the minus sign (**–**) next to the node.

You can select all object types within a category by selecting the check box at the category node level, whether or not the node is expanded. You can also select one or more individual object types within a category, after you expand the category node, by:

- Selecting the category node check box, which selects all object types within that category, and clearing the check box for each object type you want to exclude
- Clearing the category node check box, which clears all object types within that category, and selecting the check box for each object type you want to include

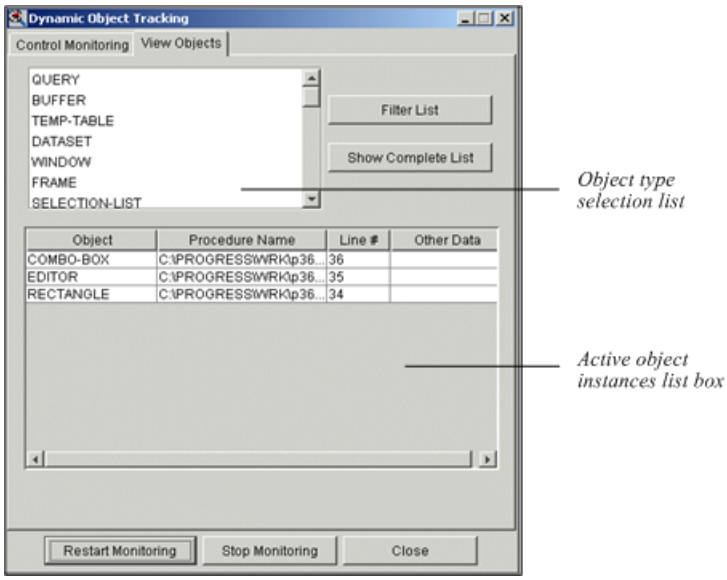
To begin tracking the creation and destruction of the object instances for the object types you selected, click the **Start Monitoring** button. Then, use the **View Objects** tab to monitor the active object instances. An active object instance is an instance of a dynamic object the application has created, but not yet destroyed, since you started or restarted monitoring.

## View Objects tab

The **View Objects** tab lists the active dynamic object instances based on the object types you selected in the **Control Monitoring** tab. The **View objects** includes the line of code where the object comes into being. The list is in order of object creation. Use this list to quickly determine what object needs to be deleted.

Active object instances are instances of dynamic objects the application has created, but not yet destroyed, since you started or restarted monitoring. The following figure shows the **View Objects** tab.

**Figure 22: Dynamic Object Tracking dialog box with View Objects tab**




While the Debugger is monitoring active object instances, the following information is displayed in the active object instances list box:

- The object type.
- The name of the procedure file in which the dynamic object was created.
- The line number at which the object was created. This is based on the Debugger listing file.
- Any relevant attribute values (such as the value of the object's `Name` attribute) or other object-specific information.

If an active object instance entry for any of the following object types is marked as "Delete Pending," execution of the `DELETE OBJECT` statement for that object instance has been delayed for some reason:

- `TEMP-TABLE` (for example, an `OUTPUT TABLE-HANDLE` parameter)
- `BUFFER` (for example, a buffer associated with an `OUTPUT TABLE-HANDLE` parameter)
- `PROCEDURE` (for example, a procedure that deletes itself using the `THIS-PROCEDURE` system handle)

  Each time the application creates a dynamic object instance, the Debugger refreshes the list of active object instances by:

  - Adding the new object instance entry (to the top of the list)
  - Removing entries for any deleted object instances

Only objects of the types you selected in the **Control Monitoring** tab appear in the object type selection list. You can filter the object instances that appear in the active object instances list box by selecting one or more object types in the object type selection list and choosing the **Filter List** button. Then, only instances of the types you select appear in the list. Use the **Show Complete List** button to cancel any filter settings and display the complete list of objects.

While monitoring, you can clear the current list of active object instances and restart monitoring in one of the following two ways:

- By choosing the **Restart Monitoring** button
- By first choosing the **Stop Monitoring** button, and then choosing the **Start Monitoring** button

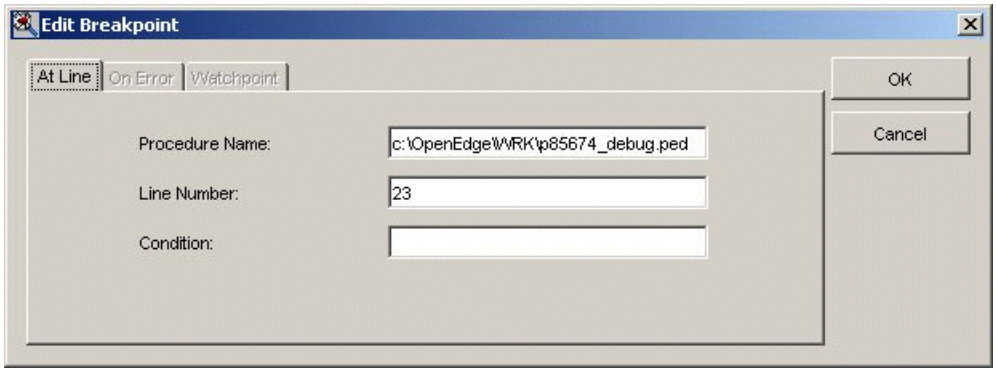
**Note:** Clearing the current list of active object instances does not destroy the actual object instances; it simply clears the current list that appears in the active object instances list box.

To monitor a different set of object types, modify your selection of object types on the **Control Monitoring** tab and click the **Start Monitoring** or **Restart Monitoring** button.

## Edit Breakpoint dialog box

The **Edit Breakpoint** dialog box lets you edit breakpoints. The following figure shows the **Edit Breakpoint** dialog box.

**Figure 23: Edit Breakpoint dialog box**




Access this dialog by choosing the **Edit** button on the **Breakpoints** dialog box. For more information, see Breakpoints dialog box on page 64.

The **Edit Breakpoint** dialog box contains the following tabs:

- **At Line** — Sets a breakpoint at a specific line in a named procedure. You can make this type of breakpoint conditional; that is, it interrupts program execution only if a defined condition is satisfied. For more information, see New Breakpoint dialog box on page 77.
- **On Error** — Sets a breakpoint to interrupt program execution if an error occurs. The error condition can be general or specific. For more information, see New Breakpoint dialog box on page 77.
- **Watchpoint** — Sets a watchpoint to interrupt program execution when the value of a variable, buffer field, or attribute reference changes. For more information, see New Breakpoint dialog box on page 77.

### Editing a breakpoint

To edit a breakpoint:

1. In the **Breakpoints** dialog box, select the breakpoint you want to edit. Click the **Edit** button.

   The breakpoint information appears in the tab of the **Edit Breakpoint** dialog box appropriate to the style of breakpoint selected. The other tabs are disabled. For example, if you select an error breakpoint, it populates the **On Error** tab. The **At Line** tab and the **Watchpoint** tab are unavailable.

2. Make your edits. For example, you might want to add a condition to a procedure line breakpoint.
3. Click **OK**.

### Editing a watchpoint

To edit a watchpoint:

1. In the **Breakpoints** dialog box, select the watchpoint you want to edit. Click the **Edit** button.

   **Note:** The **Add Watch** toggle box is unchecked by default when modifying a watchpoint.

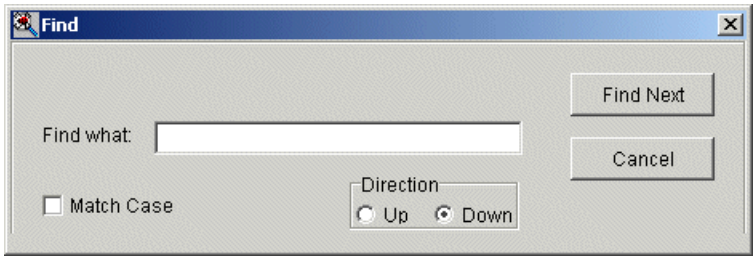
2. Modify the watchpoint expression and/or condition.
3. Click **OK** to save your changes.

   **Note:** A watchpoint remains until you explicitly delete it.

## Find dialog box

The **Find** dialog box lets you specify a text string you want to find in the source code pane. The following figure shows the **Find** dialog box.

**Figure 24: Find dialog box**




Access this dialog box by choosing **Search** > **Find** or pressing **CTRL+F**.

The **Find** dialog box also lets you specify the direction in which to search (**Up** or **Down**), and whether to match the letter case in the text string during the search (**Match Case**).

Click the **Find Next** button to begin the initial search, as well as subsequent searches, for the specified text string. The Debugger highlights the text found and scrolls the text into view, if necessary. If the text is not found, the Debugger displays an alert dialog box message.

You can also perform subsequent searches for the next and previous occurrences of the text string specified with the **Find** option by choosing **Search** > **Find Next** and **Search** > **Find Prev**, respectively. The search wraps around to the beginning of the source buffer when it reaches the end of the buffer.

## New Breakpoint dialog box

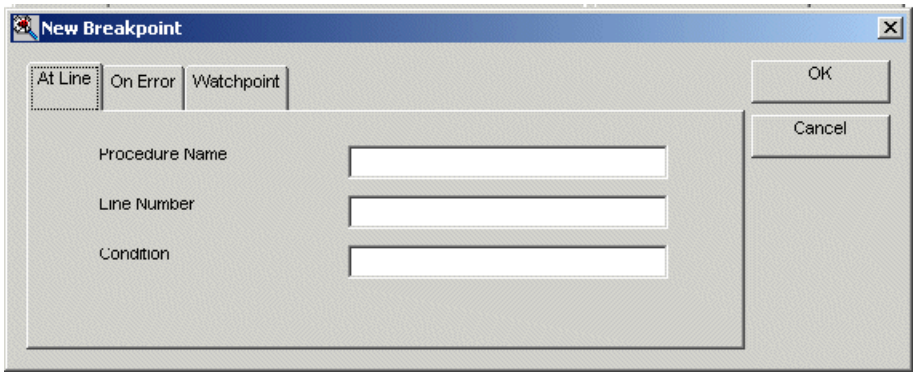
The **New Breakpoint** dialog box lets you set new breakpoints and watchpoints.

Access this dialog box by choosing the **New** button on the **Breakpoints** dialog box. For more information, see Breakpoints dialog box on page 64.

The **New Breakpoint** dialog box contains the following tabs:

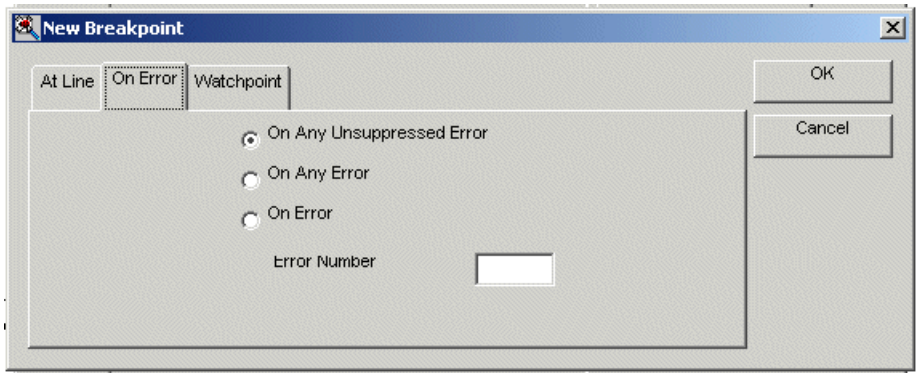
- **At Line** — Sets a breakpoint at a specific line in a named procedure. You can make this type of breakpoint conditional; that is, it interrupts program execution only if a defined condition is satisfied. A breakpoint indicator appears in red in the shaded area to the left of each line in the source code pane where an **At Line** breakpoint is set. The following figure shows the **New Breakpoint** dialog box with the **At Line** tab.

**Figure 25: New Breakpoint dialog box with At Line tab**




- **On Error** — Sets a breakpoint to interrupt program execution if an error occurs. The error condition can be general or specific. The following figure shows the **New Breakpoint** dialog box with the **On Error** tab.

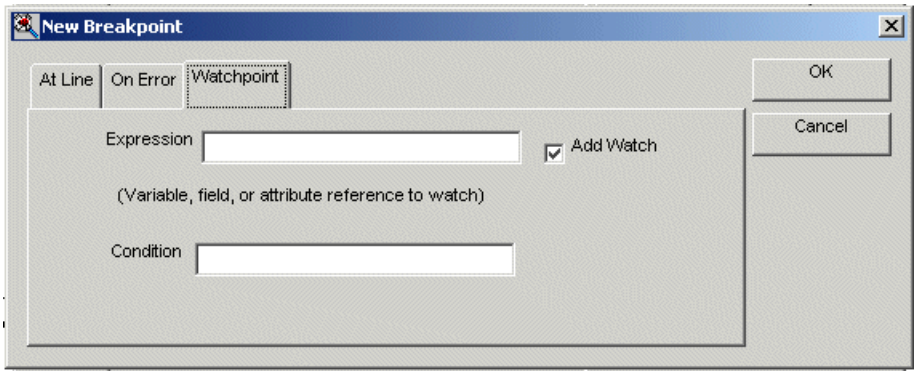
**Figure 26: New Breakpoint dialog box with On Error tab**




For more information on setting breakpoints if an error occurs, see Setting an error-based breakpoint on page 81.

- **Watchpoint** — Sets a watchpoint to interrupt program execution when the value of a variable, buffer field, or attribute reference changes. The following figure shows the **New Breakpoint** dialog box with the **Watchpoint** tab.

**Figure 27: New Breakpoint dialog box with Watchpoint tab**




## Setting a breakpoint in a procedure

You can set a breakpoint at a specific line.

To add a breakpoint at a specific line in a named procedure:

1. In the **New Breakpoint** dialog box, choose the **At Line** tab.
2. Enter the name of an ABL procedure.
3. Enter an `INTEGER` value that specifies a line number in the current or specified procedure.
4. Specify a condition that must be satisfied for the breakpoint to be honored (optional).
5. Click **OK**.

A breakpoint indicator appears in red in the shaded area to the left of each line in the source code pane where a breakpoint is set. You can also add a breakpoint by clicking in the shaded area to the left of a line in the source code pane directly.

## Procedure names and line numbers

Consider the following when setting a breakpoint:

- If you specify only a procedure name, the breakpoint is set on the first executable line of the procedure.
- If you do not specify a procedure name, the breakpoint is set on the specified line in the procedure currently displayed in the source code pane.
- If you specify a line number that is before any executable line in the code for the active procedure in the Debugger context, the breakpoint is set on the first executable line on or after the specified line. For example, if you specify line number 10 for an active procedure displayed with the following lines, the breakpoint is actually set on line 12:

```
10   /* Set Startup Temperature */
11   DEFINE VARIABLE InitialTemp AS INTEGER NO-UNDO.
12   ASSIGN InitialTemp = 32.
```

If there is no executable line on or after the specified line, the breakpoint is set on the last executable line. For example, if you specify line number 20 for the lines displayed above, the breakpoint is also set on line 12.

- If you specify a line number for an inactive procedure, the breakpoint is set on the specified line. When the procedure becomes active in the Debugger context, the specified breakpoint adjusts to the corresponding effective breakpoint line.
- When setting a breakpoint, there is a distinction between specifying a line number value less than 1 and specifying a value of 1 or more. Any value less than 1 (for example, 0 or –1) specifies the first executable line of the main procedure in the procedure file. Any positive value specifies the first executable line on or after the specified line in the procedure file. For example, suppose the procedure file starts out like this:

```
1       DEFINE VARIABLE lStart AS LOGICAL INITIAL TRUE.
2       PROCEDURE ShowStart:
3         IF lStart THEN MESSAGE "Procedure is starting ...".
4       END.
5
6       MESSAGE "Hello World!".
7       RUN ShowStart.
8       lStart = FALSE.
                      .
                      .
                      .
```

If you specify a breakpoint at line 0, –1, or any other negative value, the breakpoint actually occurs at line 6, the first line that executes in the main procedure. If you specify a breakpoint at line 1 or 2, the breakpoint occurs at line 3, the first executable line after 1 and 2, which happens to be the first executable line of an internal procedure.

This distinction also affects procedures containing the trigger phrase used to define triggers in object definitions. For example, suppose the procedure file starts out like this:

```
1       DEFINE BUTTON bOK LABEL "OK"
2         TRIGGERS:
3           ON CHOOSE
4             MESSAGE "OK Pressed!".
5         END TRIGGERS.
6       MESSAGE "Hello World!".
                     .
                     .
                    .
```

Again, if you specify a breakpoint at line –1, the breakpoint occurs on line 6, but if you specify the breakpoint at line 1, it occurs at line 4, which is the first executable line of a trigger block.

## Conditional breakpoints

A condition is any expression that the AVM can evaluate to a logical value. The expression can include variables, buffer fields, object methods and attributes, logical operators (`AND`, `OR`, and so forth), comparison operators (`=`, `<>`, `EQ`, `NE`, and so forth), and other code entities.

The AVM validates the expression syntax when the breakpoint is added, but does not check variable references. If any variables in the condition are out-of-scope or nonexistent, the AVM interrupts execution (as if honoring the breakpoint) and displays an error message in the Debugger.

You can add a condition to any existing breakpoint (except those that are triggered by an error) by selecting the breakpoint and choosing the **Edit** button in the **Breakpoints** dialog box. For more information, see Edit Breakpoint dialog box on page 76.

## Setting an error-based breakpoint

You can set a breakpoint that is triggered by the occurrence of an error condition.

To add a breakpoint triggered by an error:

1. In the **New Breakpoint** dialog box, choose the **On Error** tab. Select the appropriate radio button, as follows:
   - **Any Unsuppressed Error** — The Debugger gains control on the occurrence of any error not suppressed by a `NO-ERROR`. This type of breakpoint appears in the Breakpoints list as "Break on any unsuppressed error."
   - **Any Error** — The Debugger gains control on the occurrence of any error regardless of whether the error is suppressed by a `NO-ERROR`. This type of breakpoint appears in the Breakpoints list as "Break on any error."
   - **On Error** — The Debugger gains control on the occurrence of the specifically identified error, regardless of whether the error is suppressed by a `NO-ERROR`. This type of breakpoint appears in the Breakpoints list as "Break on Error *#nnnnn*."
2. Enter an `INTEGER` of 1–5 digits that identifies a specific error (available, and required, only when **On Error** is selected).
3. Click **OK**.

## Breakpoints triggered by errors

Consider the following when setting breakpoints based on errors:

- For any breakpoint triggered by an error, an alert box appears stating "Stopped due to Break on error *nnnnn*." (the error number appears for all three types). The Debugger gains control at the next executable line following the line of ABL code that caused the error.

---

**Note:** For **On Error**, the Debugger validates that you entered a number from 1–99999, but otherwise does not verify that an error corresponding to the number exists.

---

- Errors that occur in hidden procedures do not trigger error breakpoints. Hidden procedures are procedures whose filename begins with an underscore (_). For more information about hidden procedures, see Hidden procedure files on page 88.
- If your program does not trap a stop condition (with `ON STOP`, for example), the Debugger ignores the error that causes the interruption, since the program terminates. Similarly, the Debugger ignores an error condition that is not trapped (with `ON ERROR`, for example) and results in program termination.

## Setting a watchpoint

You can set a watchpoint to interrupt program execution when the value of a variable, buffer field, or attribute reference changes.

To add a watchpoint:

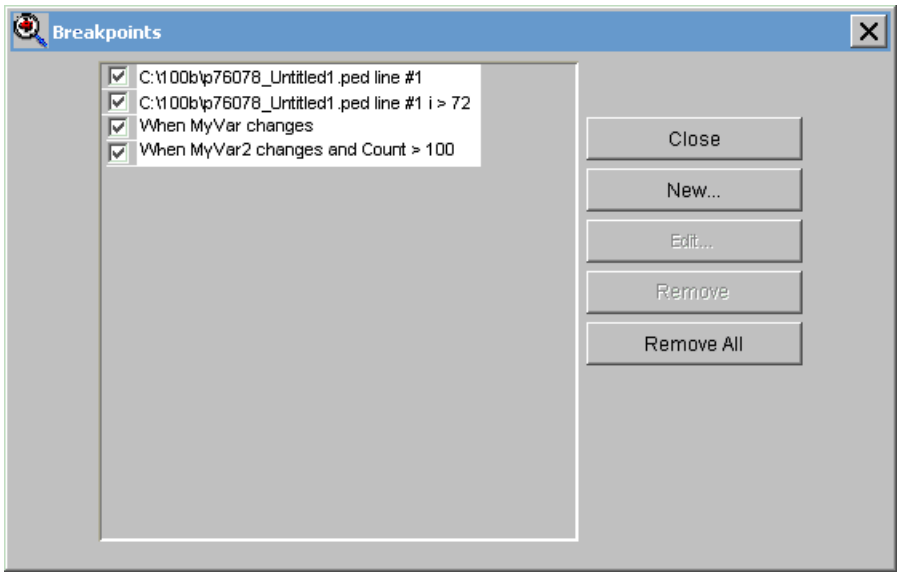
1. In the **New Breakpoint** dialog box, choose the **Watchpoint** tab.
2. Type the name of a variable, buffer field, object attribute, or object reference in the **Expression** field. You can optionally add a condition in the **Condition** field.

**Note:** A watchpoint requires a watchpoint expression. If you do not enter an expression and click OK, an error message appears in an alert box.

3. Click **OK** to create the watchpoint.

Watchpoints appear in the **Breakpoints** dialog box as "When *expression* changes [and *condition*]". The third and fourth items in the following figure are watchpoints.

**Figure 28: Watchpoint example**




Entering a watchpoint automatically creates a watch for the watchpoint expression. If you leave the **Add Watch** toggle checked, the Debugger displays the watch in the **Watches pane** of the Debugger's main window. You can uncheck the **Add Watch** toggle box to override this behavior (**Add Watch** is checked by default). The Debugger shows the current value of all watches each time the Debugger gains control of the application.

**Note:** The watch and watchpoint are not connected to each other. Deleting the watchpoint does not delete the watch. Deleting the watch does not delete the watchpoint.

If the watchpoint expression or condition is invalid, the Debugger creates the watchpoint and displays an error message. In this case, edit the watchpoint and correct the mistake. If you do not fix the watchpoint before continuing program execution, the AVM generates an error when it gains control (via **Step Into**, **Step Over**, **Continue**, etc.) and disables the watchpoint.

## Watchpoint expressions

A watchpoint expression can be a variable (including array elements), field (including array elements), attribute reference (including chained references), or built-in ABL function, such as `PROPATH`. (For a complete list of valid functions, see Table 3.) However, the expression may not be a `CLOB`, `BLOB`, `LONGCHAR`, `RAW`, or `MEMPTR` variable or field. The Debugger checks only that the watchpoint expression follows the conventions for a valid watch.

**Note:** Watchpoints are not valid on the name of an array. You must specify a particular array element to watch.

### Watchpoint conditions

A condition can be any expression that the AVM can evaluate to a logical value. The condition can include references to one or more program variables (including array elements), buffer fields (including array elements), or object attributes; you are not limited to the variable, field, or attribute specified in the watchpoint expression.

When you create a watchpoint, the Debugger checks only that the syntax of the condition is valid. The Debugger cannot verify the existence of variables and fields in the condition, as the references to variables or fields might not be in scope.

### Adding a watchpoint to a watch

To add a watchpoint to an existing watch:

Right-click on a watch in the **Watches** pane and choose **Create Watchpoint** from the pop-up menu.

The status bar displays the message "`Watchpoint created for <expression>`" to inform you that the operation was successful. If the operation fails, an alert box displays an error message.

---

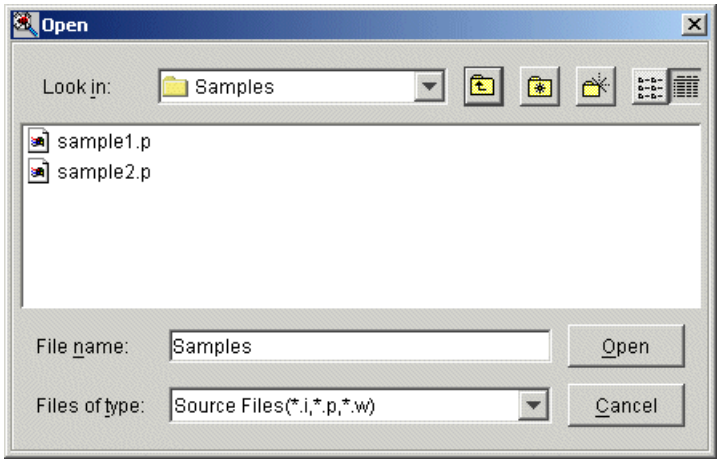
**Note:** If you want to add a condition to the new watchpoint created on an existing watch, you must bring up the **Breakpoints** dialog box, select the watchpoint, click the **Edit** button, and enter the condition in the **Edit Breakpoint** dialog box.

---

## Open dialog box

The **Open** dialog box lets you open a procedure and display it in the source code pane. The following figure shows the **Open** dialog box.

**Figure 29: Open dialog box**




Access the **Open** dialog box by choosing **File** > **Open** or pressing **F3**.

To open a procedure, select the procedure file and click the **Open** button, or double-click the procedure file. If you cannot find the document in the folder list, you can browse for it using the folder-level buttons.

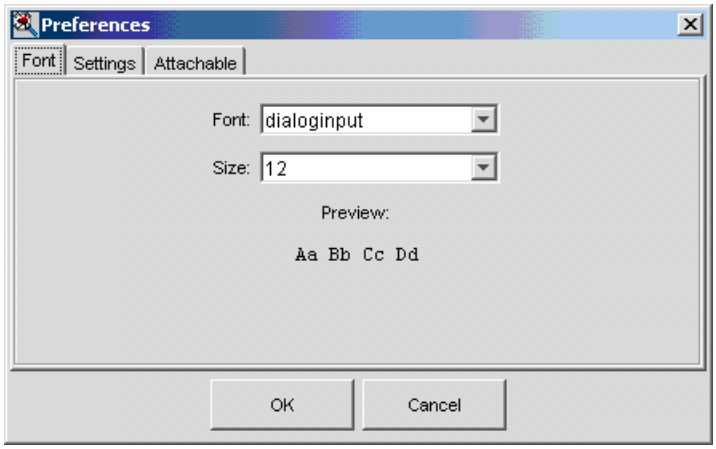
## Preferences dialog box

The **Preferences** dialog box lets you set user preferences for the Debugger window. Access this dialog box by choosing **Edit** > **Preferences**.

## Font tab

The **Font** tab sets the Debugger font and size across all panes. The following figure shows the **Font** tab.

**Figure 30: Preferences dialog box with Font tab**



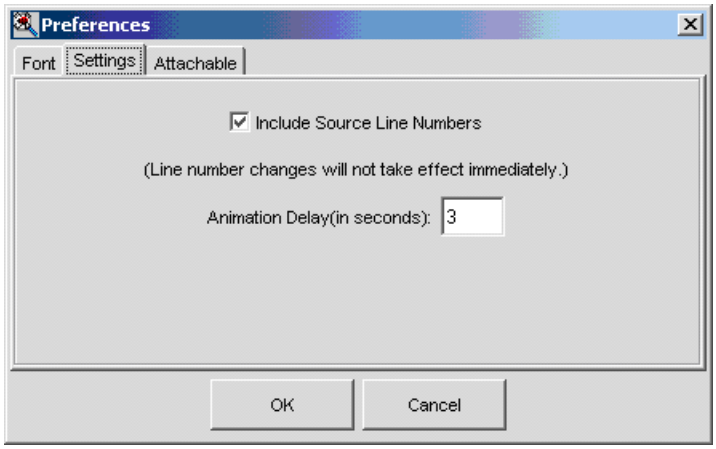

## Settings tab

The **Settings** tab sets how long to pause between steps during animated stepping and whether to include line numbers in the source code pane.

**Note:** A change to the line number setting will take effect the next time the Debugger loads a new listing file.

The following figure shows the **Settings** tab.

**Figure 31: Preferences dialog box with Settings tab**



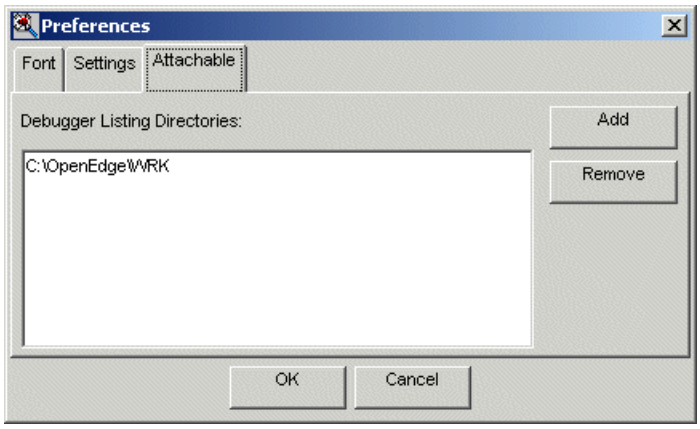

The Debugger automatically saves the following window settings:

- Debugger window size and location (set by resizing or moving the Debugger window)
- Size of the four window panes (set by resizing the panes)
- Hidden panes (set by choosing one or more options on the **View** menu)

## Attachable tab

The **Attachable** tab lists the debug listing directories. From this tab you can add and remove debug listing directories. The Debugger uses this list in attachable mode to locate the debug listing files. When an AVM instance and the Debugger are on separate machines, the debug listing directories must be accessible to the Debugger machine via a mapped drive (Windows) or a mounted file system (UNIX). The following figure shows the **Attachable** tab.

**Figure 32: Preferences dialog box with Attachable tab**




**Note:** The Debugger working directory appears by default.

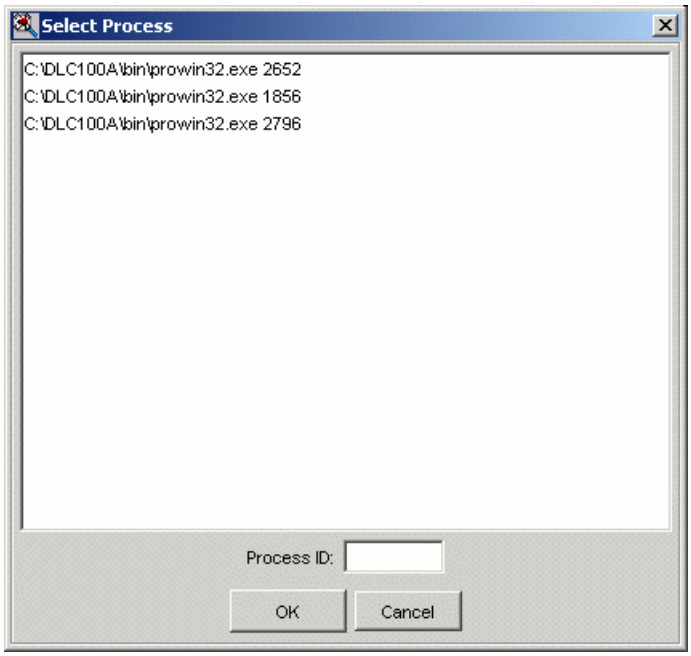
You can generate the debug listing files either manually or automatically. For more information on generating debug listing files, see Debug listing files on page 86.

The Debugger automatically opens the **Preferences** dialog box to this tab if it is unable to locate a listing file for the attached process. For more information, see Debug listing files on page 86.

## Select Process dialog box

The **Select Process** dialog box lets you obtain the list of currently running OpenEdge processes on the same machines as the Debugger. The following figure shows the **Select Process** dialog box.

**Figure 33: Select Process dialog box.**




Access this dialog box by choosing the **Get List** button on the **Attach to Process** dialog box. For more information, see Attach to Process dialog box on page 63.

To select a Process ID, locate and select the process in the list (or enter it in the **Process ID** text box), and then click **OK**.

The process ID appears in the **Process ID** text box.

# Debugger files

During a debugging session, the Debugger references procedure source files and debug listing files derived from procedure files.

The following sections describe these files, how the Debugger uses them, and how you can interact with them.

## Debug listing files

A debug listing file contains a line-numbered listing of a procedure with all preprocessor directives expanded and all include files inserted. The Debugger displays this listing in the source code pane to indicate breakpoints and the next line to execute in the current procedure.

You can generate a debug listing in two ways:

- Automatically by running a procedure during a debugging session.
- Manually by compiling a procedure explicitly using either the ABL `COMPILE` statement or the Application Compiler from the ADE Desktop.

### Automatic listing generation

Automatic listing generation is appropriate for most debugging situations where you are working with new source procedures you have not yet compiled. Automatic generation creates temporary listing files that OpenEdge removes when you end your session and exit to the operating system. These temporary listing files have alphanumeric names based on the following naming convention (without an extension):

### Syntax

```
"dbg_" + procedure-filename + "_" + 8-character unique-id
```

OpenEdge stores temporary debug listing files in either your current working directory or the temporary directory specified by the Temporary Directory (`-T`) startup parameter. For more information on how and when OpenEdge automatically generates debug listings, see Procedure files and listing generation on page 87. For more information on the Temporary Directory startup parameter, see the reference entry in *Startup Command and Parameter Reference*.

---

**Note:** There might be occasions when a session terminates leaving temporary debug listing files in your directory. This can occur during a system crash or during any other abnormal termination of your application. You can safely delete these temporary debug listings from your directory since the Debugger creates them automatically.

---

### Manual listing generation

Manual listing generation can save significant time in a debugging session, especially when debugging very large procedures. To generate a debug listing file manually, use the `DEBUG-LIST` option with the `COMPILE` statement to compile your procedure. For more information on the `COMPILE` statement and the `DEBUG-LIST` option, see *ABL Reference*.

## Locating debug listing files in attachable mode

In attachable mode, the Debugger locates listing files based on the directories set in the **Attachable tab** in the **Preferences** dialog box. The Debugger's working directory is the default listing directory. When the Debugger and AVM instance are on separate machines, the debug listing directories must be accessible to the Debugger machine via a mapped drive (Windows) or a mounted file system (UNIX). If the Debugger cannot find the listings files for any reason, it opens the **Preferences** dialog box to the **Attachable** tab so that you can provide a path. For information on this tab, see Attachable tab on page 85.

For more information on generating listing files, see Debug listing files on page 86.

## Procedure files and listing generation

You can debug procedure files in both source (`.p`, `.w`) and r-code (`.r`) form. OpenEdge makes every attempt to provide the debug listing for the procedure you are debugging. If necessary, OpenEdge generates the debug listing automatically ("on the fly"). Any automatic listing generation creates a temporary file that is deleted at the end of your session. For more information on temporary debug listings, see Debug listing files on page 86.

The method OpenEdge uses to provide a debug listing depends on the form of the procedure OpenEdge has available to execute: r-code or source code.

## Debug listing generation from r-code

When your application runs an r-code file, OpenEdge attempts to provide the debug listing for it by searching the PROPATH for the associated source file (.p, .w). The r-code contains references to all associated source, include, and debug listing files, as well as its cyclic redundancy check (CRC) value.

If OpenEdge finds the source file associated with an r-code file, it compares the CRC value of the source file with the corresponding CRC value stored in the r-code. If the CRC values do not match, OpenEdge displays a warning message and continues using the file with a bad CRC value.

You can step through r-code for which the source code is unavailable; however, the Debugger cannot display the source code in the source code pane. In this case, the Debugger displays the procedure name and line number in the status bar.

## Debug listing generation from source code

If all you have is the source code for a procedure, OpenEdge automatically generates the debug listing during compilation and saves it as a temporary file. OpenEdge generates temporary debug listings from source both when you execute a procedure for a debugging session and when you just display the procedure in the source code pane.

## Hidden procedure files

The Debugger ignores hidden procedures. A hidden procedure is a procedure whose filename begins with an underscore (_). Although you can execute a hidden procedure, it is hidden from the Debugger in the following ways:

- A hidden procedure never enters the Debugger context and does not honor any breakpoints that you set in it.
- The Debugger never displays the debug listing for a hidden procedure.

The OpenEdge ADE tools that invoke the Debugger use hidden procedures to keep the tools themselves invisible to the Debugger. You can use the same technique when writing your own development tools that invoke the Debugger.

# 4

# Using System Handles for Debugging

The DEBUGGER system handle provides attributes and methods that allow you to initialize, start, and control a debugging session from an ABL (Advanced Business Language) procedure. It is intended for experienced ABL programmers who want to:

- Control breakpoints at specific points in their code, regardless of how the surrounding code is modified, or set breakpoints in include files that might appear in many different procedures.
- Develop their own OpenEdge development tools, and initialize and control Debugger interaction from those tools. For example, the OpenEdge Application Development Environment (ADE) uses the DEBUGGER system handle to invoke and manage the Debugger from ADE tools such as the Procedure Editor and ADE Desktop.
- Debug AppServer and WebSpeed applications.

The SESSION system handle allows you to specify whether you want alert boxes to include a **Help** button that provides access to a **Stack Trace** dialog box and the Debugger.

For details, see the following topics:


- DEBUGGER system handle attributes and methods
- Using the DEBUGGER system handle to start and control the Debugger
- Using the DEBUGGER system handle in stand-alone mode
- Using the SESSION handle to start the Debugger

# DEBUGGER system handle attributes and methods

The DEBUGGER system handle attributes and methods provide a limited set of capabilities primarily designed to start up and clean up after a debugging session from an ABL procedure. The following table lists the function of each DEBUGGER system handle attribute.

**Table 19: DEBUGGER system handle attributes**

| Attribute | Function |
|---|---|
| TYPE | Returns the object type of the Debugger, the standard value for all system handles, "PSEUDO-WIDGET" |
| VISIBLE | Displays or removes the Debugger window on the screen |

The following table lists the function of each DEBUGGER system handle method.

**Table 20: DEBUGGER system handle methods**

| Method | Function |
|---|---|
| CANCEL-BREAK( ) | Cancels a specified breakpoint. |
| CLEAR( ) | Re-initializes the internal state of the Debugger. This function has no visible effect on the Debugger window. |
| DEBUG( ) | Starts and initializes the Debugger, and immediately gives control to the Debugger in stand-alone mode while blocking the invoking procedure. |
| DISPLAY-MESSAGE( ) | Displays a message in an alert box. The Debugger stores these messages and displays them to the user in an alert box when the Debugger regains control of the application. |
| INITIATE( ) | Initializes the Debugger, but does not immediately give control to the Debugger. When the ABL Virtual Machine (AVM) instance encounters a breakpoint, it gives control to the Debugger in application mode, and the procedure containing the breakpoint becomes the current procedure in the Debugger context. |
| SET-BREAK( ) | Sets a specified breakpoint. You cannot use this method to set a condition on a breakpoint, or to set a breakpoint triggered by an error. You can, however, add a condition to a breakpoint created using this method, by editing the breakpoint from the **Breakpoints** dialog box.<br>**Note:** You cannot use this method to set a a watchpoint. |

For more information on the DEBUGGER system handle or its attributes and methods, see the DEBUGGER system handle reference entry in *ABL Reference*.

You must initialize the Debugger using either the DEBUG( ) or INITIATE( ) method before using any of the remaining methods in a procedure.

**Note:** The ABL code that initiates the Debugger and displays it on the screen is responsible for removing the Debugger from the screen when it is no longer needed by setting the `DEBUGGER:VISIBLE` attribute to `FALSE`.

# Using the DEBUGGER system handle to start and control the Debugger

Any technique you use to run the Debugger from an ABL procedure runs the Debugger in either application or stand-alone mode. (For an introduction to application and stand-alone mode, see Introduction on page 21.) Each technique allows your application to interact with the Debugger context in a different way, allowing more or less control of the Debugger from your invoking application. These techniques are:

- **Application mode** — An ABL program calls the `INITIATE( )` and `SET-BREAK( )` methods. At the first breakpoint specified by the `SET-BREAK( )` method, the Debugger runs and takes control of theprocedure in which the breakpoint is set as the initial procedure in the Debugger context. In effect, the invoking procedure debugs itself. You can use this technique to implement a Debugger tool that debugs a specified subprocedure under ABL control, and optionally make the invoking procedure invisible to the Debugger context.
- **Stand-alone mode (as a tool of your application)** — An ABL program calls the `DEBUG( )` method. The invoking procedure blocks until the Debugger returns (exits). In effect, the Debugger runs as a subroutine or tool of the invoking procedure and can debug only procedures you run from the Debugger. In other words, the invoking procedure is not part of the Debugger context.

## Using the DEBUGGER system handle in application mode

This technique is useful in large applications under development where the line numbers change constantly and you want to break at specific points in your code, regardless of how the surrounding code is modified, or you want to set breakpoints in include files that appear in many different procedures. This technique is also useful when implementing a Debugger tool in application mode, where you can set a breakpoint in a called subprocedure to indicate where debugging should start. Use this technique to start the Debugger on an AppServer to debug a remote procedure. For more information, see Remote debugging on page 30.

### Debugging by controlling breakpoints from an ABL procedure

To start the Debugger and control breakpoints from an ABL procedure:

1. Optionally, add a `DEFINE VARIABLE` statement that defines a logical variable you can use to assign the return value for `DEBUGGER` system handle methods.
2. Add a `DEBUGGER` system handle statement that invokes the `INITIATE( )` method before the point where you want to begin debugging. This initializes the Debugger but does not immediately make it visible.
3. Add one or more `DEBUGGER` system handle statements that invoke the `SET-BREAK( )` method to set at least one breakpoint that occurs after the statement where you set it. The `SET-BREAK( )` method that sets the breakpoint must also execute after the statement that invokes the `INITIATE( )` method in the previous step.
4. Run the invoking procedure.

For example, in the following listing fragment, the Debugger is initialized on line 6, and the procedure stops at a breakpoint on line 15. The Debugger takes control at this point. From here, you can continue executing the invoking procedure under Debugger control, stopping at and continuing from all breakpoints, as shown:

```
        1   DEFINE VARIABLE procname
        2     AS CHARACTER NO-UNDO EXTENT 6.
        3   DEFINE VARIABLE Selection
        4     AS INTEGER   NO-UNDO FORMAT "9".
        5   DEFINE VARIABLE debug AS LOGICAL NO-UNDO.
        6   debug=DEBUGGER:INITIATE().
        7   /* Initialize the list of procedure names to be run */
        8   procname[1] = "custedit.p".
        9   procname[2] = "custrpt.p".
       10   procname[3] = "ordedit.p".
       11   procname[4] = "ordrpt.p".
       12   procname[5] = "itemedit.p".
       13   procname[6] = "itemrpt.p".
       14   debug=DEBUGGER:SET-BREAK().
* =>   15   DISPLAY "S A L E S   O R D E R   S Y S T E M" SKIP(1)
       16     WITH COLUMN 18 NO-BOX.
       17   . . .
```

When you exit the Debugger, the AVM instance continues execution from its current stopping (or breaking) point. If you exit the procedure before exiting the Debugger, the Debugger window closes and control returns to the startup environment (such as the Procedure Editor or AppBuilder).

## Debugging a called subprocedure

To debug a subprocedure called from the invoking procedure:

1. Optionally, add a `DEFINE VARIABLE` statement that defines a logical variable you can use to assign the return value for `DEBUGGER` system handle methods.
2. Add a `DEBUGGER` system handle statement that invokes the `INITIATE( )` method before the point where you want to begin debugging the called subprocedure. This initializes the Debugger but does not immediately make it visible.
3. Add a `DEBUGGER` system handle statement that invokes the `SET-BREAK( )` method to set a breakpoint at the first executable line of the subprocedure you want to debug. This statement must execute after the statement that invokes the `INITIATE( )` method in the previous step, but before the statement that calls the subprocedure.
4. Run the invoking procedure.

The following listing fragment prompts the user to enter the name (procname) of a procedure to debug (line 30). It then initializes the Debugger (line 33) and sets a breakpoint on the first executable line of procname (line 34). When the invoking procedure calls procname (line 35), procname breaks immediately on its first executable line and gives control to the Debugger. The Debugger then displays the listing for procname in the Debugger window as the current procedure, as shown:

```
 1   DEFINE VARIABLE debug    AS LOGICAL   NO-UNDO.
 2   DEFINE VARIABLE procname AS CHARACTER NO-UNDO.
     . . .
30   SET procname LABEL "Enter name of procedure to debug" DEBLANK
31     WITH SIDE-LABELS.
32   IF procname <> "" THEN DO:
33     debug = DEBUGGER:INITIATE().
34     debug = DEBUGGER:SET-BREAK(procname).
35     RUN VALUE(procname).
36   END.
     . . .
```

**Note:** Using this technique, the invoking procedure becomes part of the Debugger context; thus, the invoking procedure appears on the call stack. You can set breakpoints in the invoking procedure and break into it with the Debugger. In this type of application, you might not want the invoking procedure running in the Debugger context like this. You can prevent the invoking procedure from entering the Debugger context by prepending its procedure name with an underscore (_). In this case, the Debugger ignores the invoking procedure completely, and it never appears on the call stack.

Although this is a simple example, it illustrates the basic elements required to build an OpenEdge development tool that can start debugging sessions for specified application procedures.

## Using the DEBUGGER system handle in stand-alone mode

This technique is useful for running the Debugger as a tool started from one application to debug other applications. The invoking procedure runs the Debugger much like a subroutine, but it does not specify any procedure for the Debugger to run as its initial current procedure. You must run all procedures you want to debug from the Debugger after it starts by using the Run option on the Debug menu.

To start the Debugger from an ABL procedure while blocking the invoking procedure:

1. Optionally, add a DEFINE VARIABLE statement that defines a logical variable you can use to assign the return value for the DEBUGGER system handle method.
2. Add a DEBUGGER system handle statement that invokes the DEBUG( ) method at the point where you want to start the Debugger.

In the following example, the procedure presents four execution options. The first three options invoke procedures. The fourth option invokes the `DEBUG( )` method and blocks at the subsequent statement (line 33) while the Debugger runs, as shown:

```
        1   DEFINE VARIABLE procname   AS CHARACTER NO-UNDO EXTENT 3.
        2   DEFINE VARIABLE programNum AS INTEGER   NO-UNDO INITIAL 1
        3      VIEW-AS RADIO-SET
        4      RADIO-BUTTONS "Customer Report", 1,
        5                    "Order Report", 2,
        6                    "Item Report", 3,
        7                    "Other", 4.
        8   DEFINE BUTTON runReport LABEL "Run Report".
        9   DEFINE BUTTON done LABEL "Done".
       10
       11   /* Initialize the list of procedure names to be run */
       12   procname[1] = "custrpt.p".
       13   procname[2] = "ordrpt.p".
       14   procname[3] = "itemrpt.p".
       15
       16   DISPLAY "R E P O R T I N G   S Y S T E M" SKIP(1)
       17            WITH COLUMN 18 NO-BOX.
       18     FORM
       19       SKIP(1) space(1) programNum
       20       SKIP(1) SPACE(1) runReport SPACE(2) done SPACE(1)
       21       SKIP(1)
       22       WITH FRAME mainMenu
       23       COLUMN 21 NO-LABELS
       24       TITLE "M A I N   M E N U".
       25
       26     ON 'choose':U OF runReport IN FRAME mainMenu
       27     DO:
       28       ASSIGN programNum.
       29       IF programNum < 4 THEN
       30           RUN VALUE(procname[programNum]).
       31       ELSE
       32           DEBUGGER:DEBUG().
=>     33     END.
       34
       35   ON 'choose':U OF done IN FRAME mainMenu
       36       APPLY "close" TO THIS-PROCEDURE.
       37
       38   ENABLE ALL WITH FRAME mainMenu.
       39   WAIT-FOR CLOSE OF THIS-PROCEDURE.
```

When the Debugger starts, you can open a procedure using **File** > **Open** and then run the procedure using **Debug** > **Run**.

As a result, while the invoking procedure waits on the blocked statement, the Debugger runs on its own, allowing you to control any other procedure that you start from it.

When you exit the Debugger, it returns control to the invoking procedure, which continues execution from the next statement (line 33 in the example). In stand-alone mode, you cannot otherwise access the invoking procedure until the Debugger returns control.

## Using the SESSION handle to start the Debugger

If you want alert boxes to include a **Help** button that provides access to a **Stack Trace** dialog box and the Debugger, use the `SESSION` handle and set the `DEBUG-ALERT` attribute to `TRUE`. Use the following sample code if you want alert boxes to include a **Help** button:

```
SESSION:DEBUG-ALERT = TRUE.
```

For more information on starting the Debugger from alert boxes, see Starting and Stopping the Debugger on page 33

**Note:** You can enable this feature when you start an ABL client using the Debug Alert (`-debugalert`) startup parameter. For information on this startup parameter, see the reference entry in *Startup Command and Parameter Reference*.

II

# Troubleshooting Resources

For details, see the following topics:

- Logging in OpenEdge
- Log Entry Types Detail
- Other Diagnostics Resources

# 5

# Logging in OpenEdge

The OpenEdge logging infrastructure provides a mechanism for logging diagnostic information based on a set of logging characteristics.

For details, see the following topics:

- Logging infrastructure overview
- Log files
- Log size threshold
- Log entry types
- Logging levels
- Logging settings

## Logging infrastructure overview

The logging infrastructure provides a means for either standard reporting of run-time activity or logging diagnostic data for troubleshooting problems based on a set of logging characteristics. OpenEdge writes information to a log file. You control the type and amount of information logged using a combination of log entry types and logging levels. Log entry types are categories of information written to the log file. Logging levels control the amount of logging information for any given log entry type and environment. Log files are described in Log files on page 104. Log entry types are described in Log entry types on page 110. Logging levels are described in Logging levels on page 117.

The logging infrastructure is available for all ABL (Advanced Business Language) clients, as well as the AIA, NameServer, Unified Broker, and DataServer components. Each of these components use a particular set of characteristics to specify logging within the infrastructure. The following table shows how to specify logging characteristics for each of these components.

**Table 21: Specifying component logging characteristics**

| For . . . | Specify settings via . . . |
|---|---|
| GUI or character client | Startup Parameters `LOG-MANAGER` system handle |
| Run-time client | `LOG-MANAGER` system handle |
| AppServer or WebSpeed agent | OpenEdge Management or OpenEdge Explorer `LOG-MANAGER` system handle |
| WebSpeed Messenger | OpenEdge Management or OpenEdge Explorer |
| Unified Broker | OpenEdge Management or OpenEdge Explorer |
| NameServer | OpenEdge Management or OpenEdge Explorer |
| AIA | OpenEdge Management or OpenEdge Explorer |
| DataServer | Startup ParametersOpenEdge Management or OpenEdge Explorer `LOG-MANAGER` system handle `DSLOG-MANAGER` system handle |

## LOG-MANAGER system handle attributes and methods

The `LOG-MANAGER` system handle attributes and methods provide a limited set of capabilities to access the logging settings for the current ABL runtime session.

The following table lists the function of each `LOG-MANAGER` system handle attribute.

**Table 22: LOG-MANAGER system handle attributes**

| Attribute | Access | Function |
|---|---|---|
| ENTRY-TYPES-LIST | Readable | Returns a character string containing a comma-separated list of all valid entry types for the current OpenEdge environment. |
| LOG-ENTRY-TYPES | Readable/Writable | Specifies a comma-separated list of one or more types of log entries to write to the log file. |
| LOGFILE-NAME | Readable/Writable | Specifies the name of log file OpenEdge uses to log messages and ABL stack trace information. |
| LOGGING-LEVEL | Readable/Writable | Specifies the level at which log entries are written to the log file. |
| LOG-THRESHOLD | Readable | Returns the file size threshold of log files. When the current log file becomes equal to or greater than the specified size, OpenEdge renames and saves the log file and creates a new log file. |
| NUM-LOG-FILES | Readable | Returns the number of rolled over log files to keep on disk at any one time, for OpenEdge session, including the current log file. |
| TYPE | Readable | Returns the type of handle (for all system handles, "PSEUDO-WIDGET"). |

The following table lists the function of each LOG-MANAGER system handle method.

**Table 23: LOG-MANAGER system handle methods**

| Method | Function |
|---|---|
| CLEAR-LOG( ) | Clears all messages existing in the current client log file and leaves the file open for writing |
| CLOSE-LOG( ) | Stops an interactive or batch client from writing messages to the current log file, and closes the log file |
| WRITE-MESSAGE( ) | Writes user messages to the current log file |

**Note:** The CLOSE-LOG( ) method is valid only for interactive and batch clients. WebSpeed agents and AppServer servers write a message to the server log file indicating that it is invalid to use the CLOSE-LOG( ) method to close a WebSpeed or AppServer server log file. DataServer connections established by Webspeed or AppServer agents all share the same server log file. So, you cannot execute the CLOSE-LOG( ) method unless there is only one agent with an open DataServer connection. In this case, the method returns FALSE.

For more information on the LOG-MANAGER system handle or its attributes and method, see the LOG-MANAGER system handle reference entry in *ABL Reference.*

## DSLOG-MANAGER system handle attributes and methods

The DSLOG-MANAGER system handle attributes and methods provide a limited set of capabilities to access the logging settings for the DataServer's server context.

**Note:** The DSLOG-MANAGER is implemented for the DataServer for MS SQL Server and the DataServer for Oracle. For runtime access to logging setting for the DataServer's client context, use the LOG-MANAGER system handle from the current ABL runtime session as described in the previous section.

A user can have a valid LOG-MANAGER and DSLOG-MANAGER system handle at the same time in an OpenEdge application since they control different log contexts. The LOG-MANAGER manages the client (or server, in the case of AppServer/WebSpeed) log file and settings which are global to any given ABL process. For DataServers, each connection to a DataServer is its own entity. For instance, you may have 2 MS SQL Server databases connected (or of other types, such as ORACLE) and each connection works independent of the other. Each DataServer connection is a different server context and may have different logging settings. In order to support the runtime management of log settings for each connected database independently, the DSLOG-MANAGER provides the DB-CONTEXT attribute. By default, it is the unknown value which disassociates the system handle with any particular connection. The user can set the system handle to the logical database name (or alias name) of an existing connection in order to apply log settings. At another time, the user can change the DB-CONTEXT attribute back to the unknown value or to the logical database name (or alias) of another existing connection in order to modify settings for that alternative connection.

**Note:** The supported DB-TYPES for the DSLOG-MANAGER system handle are MSS and ORACLE. If the database does not have the correct type, or is not connected, or has an invalid value, an error is raised by the system handle.

The following table lists the function of each DSLOG-MANAGER system handle attribute.

**Table 24: DSLOG-MANAGER system handle attributes**

| Attribute | Access | Function |
|---|---|---|
| DB-CONTEXT | Readable/Writable | Attribute should be set with the name of a valid and connected logical database name of a supported db-type before any of the other attributes and method of DSLOG-MANAGER can be accessed. You can also set it to the alias of a connected database. Default value is the unknown value. When you read the DB-CONTEXT attribute, it returns the same value as it was set to unless you specify a database alias, in which case it returns the logical name of the database assigned to the alias at the time the attribute was set. |
| ENTRY-TYPES-LIST | Readable | Returns a character string containing a comma-separated list of all valid entry types for the current OpenEdge environment. |
| LOG-ENTRY-TYPES | Readable/Writable | Specifies a comma-separated list of one or more types of log entries to write to the log file. |
| LOGFILE-NAME | Readable | Specifies the name of log file OpenEdge uses to log messages and ABL stack trace information. LOGFILE-NAME is read-only because the DataServer cannot currently work without a log file opened. This disallows the changing of the log file at runtime and avoids the risk of having no log file opened if the file open on a 'new' log file should fail. |
| LOGGING-LEVEL | Readable/Writable | Specifies the level at which log entries are written to the log file. |
| LOG-THRESHOLD | Readable | Returns the file size threshold of log files. When the current log file becomes equal to or greater than the specified size, OpenEdge renames and saves the log file and creates a new log file. |
| NUM-LOG-FILES | Readable | Returns the number of rolled over log files to keep on disk at any one time, for OpenEdge session, including the current log file. |
| TYPE | Readable | Returns the type of handle (for all system handles, "PSEUDO-WIDGET"). |

The following table lists the function of each DSLOG-MANAGER system handle method.

**Table 25: DSLOG-MANAGER system handle methods**

| Method | Function |
|---|---|
| `CLEAR-LOG( )` | Clears all messages existing in the current client log file and leaves the file open for writing |
| `WRITE-MESSAGE( )` | Writes user messages to the current log file |

**Note:** The `CLOSE-LOG( )` is not a supported `DSLOG-MANAGER` system handle method. You cannot execute the `CLOSE-LOG( )` method unless there is only one brokered DataServer server spawned by the Unified Broker or by ProBroker.

For more information on the `DSLOG-MANAGER` system handle or its attributes and methods, see the `DSLOG-MANAGER` system handle reference entry in the *Use the Microsoft SQL Data Server*, and *Use the Oracle Data Server*.

# Log files

A log file contains log message header information and log message content. The message content is set by the log entry type and the logging level. Log entry types are described in Log entry types on page 110. Logging levels are described in Logging settings on page 119.

The message header information is consistent across all combinations of log entry types and logging levels. For example:

```
[06/10/31@14:07:15.892-0400] P-002702 T-002867 2 4GL DYNOBJECTS
```

Where:

- 06/10/31 is the date in *yy/mm/dd* format
- 14:07:15.892 is the time in *hh:mm:ss.mls*
- 0400 is the time zone, expressed as the number of hours relative to GMT
- 002702 is the process ID
- 002867 is the thread ID
- 2 is the logging level associated with this log entry
- 4GL is the execution environment
- DYNOBJECTS is the log entry type

For an ABL client, specify the log file with the Client Logging (-clientlog) startup parameter. For AppServer and WebSpeed, the log is the normal AppServer or WebSpeed log file (for example, asbroker1.server.log). For WebSpeed Messenger, Unified Broker, NameServer, AIA, the log file is specified through OpenEdge Management or OpenEdge Explorer. For DataServers, specify the log file for the client content with the Client Logging (-clientlog) startup parameter; specify the log file for the server content with the DataServer Logging (-dslog) startup parameters.

## Specifying the log filename

You can specify the name of the log file to which OpenEdge writes all errors and warning messages by using the following:

- The Client Logging (-clientlog) startup parameter
- The LOGFILE-NAME attribute on the LOG-MANAGER system handle
- The logFile and related properties in the ubroker.properties file with OpenEdge Management or OpenEdge Explorer

**Note:** Use the above to log specifications to designate log file information for DataServer clients as well. The server context of DataServers, however, starts with a default log file name of dataserver.lg and uses separate mechanisms for designating a log file name. See the DataServer Guide for MS SQL Server for more information about specifying a log filename for the server context.

The following table shows how to specify the log filename.

**Table 26: Specifying the log filename**

| Method | General syntax | Example |
|---|---|---|
| Startup parameter | -clientlog *filename* | -clientlog mylog.log |
| LOG-MANAGER system handle | LOG-MANAGER:LOGFILE-NAME = "*filename*" | LOG-MANAGER:LOGFILE-NAME = "mylog.log" |
| OpenEdge Management or OpenEdge Explorer, or ubroker.properties files | logFile = *filename*srvrLogFile= *filename*brkrLogFile= *filename* | logFile = c:\workdir\mylog.log |

**Note:** After starting an ABL session using the Client Logging (-clientlog) startup parameter, another log filename can be specified in code using the LOGFILE-NAME attribute on the LOG-MANAGER system handle. In this case, OpenEdge stops writing to the log file specified by the Client Logging (-clientlog) startup parameter and starts writing to the log file specified by the LOGFILE-NAME attribute.

For a detailed description of the Client Logging (-clientlog) startup parameter, see *Startup Command and Parameter Reference*. For a detailed description of the LOG-MANAGER system handle, see *ABL Reference*.

For more information on ubroker.properties, see the ubroker.properties.README file in $INSTALLDIR/properties/ubroker.properties.README.

To use OpenEdge Management or OpenEdge Explorer to view and set properties, see Logging settings on page 119.

## Clearing the log file

You can clear messages from the current log file and leave the log file open for writing using the following methods:

- The Clear Log (-clearlog) startup parameter
- The CLEAR-LOG( ) method on the LOG-MANAGER system handle

**Note:** This method is valid only for interactive or batch clients.

- The srvrLogAppend and related properties in the ubroker.properties file with OpenEdge Management or OpenEdge Explorer

The following table shows how to clear the log file.

**Table 27: Clearing the log file**

| Method | General syntax | Example |
|---|---|---|
| Startup parameter | -clearlog | -clientlog mylog.log -clearlog |
| LOG-MANAGER system handle | LOG-MANAGER:CLEAR-LOG( ) | LOG-MANAGER:CLEAR-LOG( ) |
| OpenEdge Management or OpenEdge Explorer, or ubroker.properties files | logAppend = 0 \| 1<br>srvrLogAppendFile=0 \| 1<br>brkrLogAppend= 0 \| 1 | srvrLogAppendFile=0 |

For a detailed description of the Clear Log (-clearlog) startup parameter, see *Startup Command and Parameter Reference*.

For a detailed description of the LOG-MANAGER system handle, see *ABL Reference*.

## Writing user messages to the log file

You can write user messages to the current log file by using the WRITE-MESSAGE( ) method on the LOG-MANAGER system handle.

**Note:** For interactive or batch clients, the WRITE-MESSAGE( ) method writes the log entries to the log file specified by the LOGFILE-NAME attribute or the Client Logging (-clientlog) startup parameter. For WebSpeed agents and AppServer agents, the WRITE-MESSAGE( ) method writes the log entries to the server log file.

The following table shows how to write messages to the log file.

**Table 28: Writing messages to the log file**

| Method | General syntax | Example |
|---|---|---|
| LOG-MANAGER system handle | LOG-MANAGER:WRITE-MESSAGE( *msg-exp* [ , *subsys-expr* ] ) | LOG-MANAGER:WRITE-MESSAGE("Got here: x=" + string(x), "DEBUG1"). |

The following line appears in the log file:

```
[06/10/31@13:19:19.742-0500] P-003616 T-001984 1 4GL DEBUG1    Got here: x=5
```

For a detailed description of the LOG-MANAGER system handle, see *ABL Reference*.

## Closing the log file

You can stop writing messages to the current log file, and close the current log file by using the CLOSE-LOG( ) method on the LOG-MANAGER system handle.

**Note:** This method is only valid for interactive or batch clients. For WebSpeed or AppServer agents, the log file closes when the broker or server shuts down.

The following table shows how to close the log file.

**Table 29: Closing the log file**

| Method | General syntax | Example |
|---|---|---|
| LOG-MANAGER system handle | LOG-MANAGER:CLOSE-LOG( ) | LOG-MANAGER:CLOSE-LOG( ) |

For a detailed description of the LOG-MANAGER system handle, see *ABL Reference*. This method is not available to the server context of DataServers via the DSLOG-MANAGER system handle due to the multi-user nature of the open log file for DataServers

# Log size threshold

The log file has a size threshold, allowing more control over the amount of disk space used for log files. When the file becomes equal to or greater than the specified size, the client process renames it and creates a new log file. The default value is 0, which implies there is no limit. Possible Values are 500000–2147483647 (in bytes).

**Note:** The log size threshold setting of 0 implies to ignore the number of log files setting.

To specify the number of rolled-over log files, see Specifying the number of log files retained on page 108.

## Specifying the log size threshold

You can specify the log size threshold by using:

- The Log Threshold (`-logthreshold`) startup parameter
- The `logThreshold` and related properties in the `ubroker.properties` file with OpenEdge Management or OpenEdge Explorer

The following table shows how to specify the log size threshold.

**Table 30: Specifying the logsize threshold**

| Method | General syntax | Example |
|---|---|---|
| Startup parameter | `-logthreshold`*n* | `-logthreshold 2000000` |
| OpenEdge Management or OpenEdge Explorer, or `ubroker.properties` files | `logThreshold=` *n* `srvrLogThreshold =` *n* `brkrLogThreshold =` *n* | `logThreshold= 2000000` |

**Note:** You can read the log size threshold by using the `LOG-THRESHOLD` attribute on the `LOG-MANAGER` system handle.

For a detailed description of the Log Threshold (`-logthreshold`) startup parameter, see *Startup Command and Parameter Reference*.

For a detailed description of the `LOG-MANAGER` system handle, see *ABL Reference*.

For more information on `ubroker.properties`, see the `ubroker.properties.README` file in `$INSTALLDIR/properties/ubroker.properties.README`.

To use OpenEdge Management or OpenEdge Explorer to view and set properties, see Logging settings on page 119.

This attribute is read only for the `DSLOG-MANAGER` system handle due to the multi-user nature of the open log file for DataServers. This value can only be set at client or broker startup.

## Specifying the number of log files retained

You can specify how many rolled-over log files to keep when a log threshold value is set for a given log file. The specified number represents the maximum total number of log files to keep on disk at any time, including the log currently being written to. The number takes into account any log files that already exist at the time the session starts up plus any new ones created during that session. For example, if there are five log files on the disk at startup and you specify you want a maximum of three, OpenEdge deletes the oldest two. Then, each time it creates a new one, OpenEdge deletes the next oldest one. The default value is 3. Possible values are 0, 2, 3 or greater.

**Note:** A value of 0 means that OpenEdge will create as many files as necessary. Values of 1 or greater than 999999 are not valid and cause a run-time error to occur and the OpenEdge process terminates.

You can specify the number of log files retained by using:

- The Number of Log Files to Keep (-numlogfiles) startup parameter
- The numLogFiles and related properties in the ubroker.properties file with OpenEdge Management or OpenEdge Explorer

The following table shows how to specify the number of log files retained.

**Table 31: Specifying the number of retained log files**

| Method | General syntax | Example |
|---|---|---|
| Startup parameter | -numlogfiles*n* | -numlogfiles 2 |
| OpenEdge Management or OpenEdge Explorer, or ubroker.properties files | numLogFiles = *n*srvrNumLogFiles = *n*brkrNumLogFiles = *n* | numLogFiles = 2 |

**Note:** You can read the number of log files retained by using the NUM-LOG-FILES attribute on the LOG-MANAGER system handle.

For a detailed description of the Number of Log Files to Keep (-numlogfiles) startup parameter, see *Startup Command and Parameter Reference*.

For a detailed description of the LOG-MANAGER system handle, see *ABL Reference*.

For more information on ubroker.properties, see the ubroker.properties.README file in $INSTALLDIR/properties/ubroker.properties.README.

To use OpenEdge Management or OpenEdge Explorer to view and set properties, see Logging settings on page 119.

This attribute is read only for the DSLOG-MANAGER system handle due to the multi-user nature of the open log file for DataServers. This value can only be set at client or broker startup.

## Filename for rolled-over log files

To order log files and avoid name conflicts, OpenEdge uses the following format for naming rolled-over log files:

```
filename.######.extension
```

*filename*

Log filename you specify.

*######*

OpenEdge adds the sequence number starting at 000001 and increases for each new log file until 999999, after which it rolls back to 000001.

*extension*

Log file extension you specify.

In the following example the log file name is `asbroker1.server` and the log file extension is `log`:

```
asbroker1.server.000001.log
```

**Caution:** Do not specify the filename with a sequence number in the Client Logging (`-clientlog`) startup parameter or for the related property (in the `ubroker.properties` file). OpenEdge determines the correct filename and sequence number, based on the filename you specify.

### srvrLogWatchDogInterval property

For AppServer and WebSpeed environments, there also is a property for the Unified Broker, named `srvrLogWatchDogInterval`, which specifies the number of seconds to wait before checking to see if the log file is full. The watchdog thread checks the server log file size every *n* seconds to verify if it needs to roll over the file. The default value is `60` (1 minute). This property takes effect only if `srvrLogthreshold` is specified. If you decrease or increase this value, the log file rolls over when its size is closer to or further from the specified threshold value. In other words, the more frequently the log is checked, the closer its size will be to the threshold value. And consequently, the greater the time between checks, the larger the file is likely to be, relative to the threshold value.

## Log entry types

Log entry types represent categories of information you can write to a log file. You can categorize the information into server and client contexts. The log entry types in the client context are applicable to DataServer clients, ABL clients, AppServer agents and WebSpeed agents. However, the log entry types in the server context are applicable only to DataServer brokers.

The degree of detail written to the log file is determined by the logging level. For more information on logging levels, see Logging levels on page 117.

### Log entry types (Client context)

The following table lists the available log entry types in the client context.

**Table 32: Log entry types (Client context)**

| Log entry type | Executable | Description |
|---|---|---|
| 4GLMessages | ABL (GUI and character mode); AppServer and WebSpeed do not require this log entry type for ABL messages to be written to the log file | Turns on logging of ABL messages. OpenEdge writes all ABL VIEW-AS ALERT-BOX messages to the log file, together with the ABL stack, when you turn on Debug Alert, using either the Debug Alert (-debugalert) startup parameter or the DEBUG-ALERT attribute on the SESSION system handle<br>.For log entry details, see the ABL message logging. |
| 4GLTrace | ABL clients, AppServer, and WebSpeed agents | Turns on logging for the execution of internal procedures, user-defined functions, persistent user-interface triggers, named events, and class events (generated by the RUN, FUNCTION, PUBLISH, and SUBSCRIBE statements, and (for class events) by the Publish( ), Subscribe( ), and Unsubscribe( ) event methods, respectively). It also logs the instantiation and use of classes, including execution of constructors (invoked by the NEW phrase and also by the SUPER and THIS-OBJECT statements), the execution of methods defined within classes (including those invoked using the SUPER system reference), the execution of property accessor methods (invoked by accessing a property of a class), and the execution of destructors (invoked by executing the DELETE OBJECT statement).<br>For log entry details, see ABL trace logging on page 122. |
| 4GLTrans | ABL clients | Turns on logging for the processing of transactions and subtransactions in ABL procedures.<br>For log entry details, see ABL transaction logging on page 124. |

| Log entry type | Executable | Description |
|---|---|---|
| `AiaDefault`<br>`AiaMgmt`<br>`AiaProp`<br>`AiaRqst`<br>`AiaUbroker` | AIA | Turns on logging for the AIA component.<br>For log entry details, see AIA logging on page 126. |
| `ASDefault` | AppServer agent | Combines the `ASPlumbing` and `DB.Connects` log entry types. It is the default value for AppServer agents.<br>For log entry details, see AppServer logging on page 127. |
| `ASPlumbing` | AppServer agent | Turns on logging for different actions, depending on the logging level specified.<br>For log entry details, see AppServer logging on page 127. |
| `DB.Connects` | ABL clients, AppServer, and WebSpeed agents | Turns on logging of database connections (connects and disconnects). The log messages include database name and user ID number.<br>For log entry details, see Database connection logging on page 128. |
| `DS.Cursor` | DataServer clients | Turns on logging for cursor identification from the OpenEdge client and shows various attributes tagged to the cursor selection.<br>For log entry details, see LOG-MANAGER system handle attributes and methods on page 100. |
| `DS.QryInfo` | DataServer clients | Turns on logging for query preparation from the OpenEdge client and shows various attributes tagged to server query execution.<br>For log entry details, see LOG-MANAGER system handle attributes and methods on page 100. |

| Log entry type | Executable | Description |
|---|---|---|
| `DynObjects.UI`<br>`DynObjects.Class`<br>`DynObjects.DB`<br>`DynObjects.Other`<br>`DynObjects.XML` | ABL clients, AppServer, and WebSpeed agents ABL clients and WebSpeed agents | Turns on logging of dynamic object creation and deletion.<br>For log entry details, see Dynamic object logging on page 131. |
| `FileID` | ABL clients, AppServer, and WebSpeed agents | Turns on logging of file access operations (opening and closing files), and any file access error messages that might occur.<br>For log entry details, see File access logging on page 149. |
| `IgnoredOps` | ABL clients | Turns on logging of operations that the AVM ignores because the ABL windows to which they apply are embedded in .NET forms. For information about operations that the AVM ignores when they occur in the context of an embedded window, see the EmbeddedWindow property reference entry in *ABL Reference*.<br>For log entry details, see Ignored operation logging on page 150. |
| `MsgrTrace` | WebSpeed Messengers | Turns on logging for WebSpeed Messengers. The information logged depends on which Messenger is running and the logging level specified.<br>For log entry details, see WebSpeed Messenger logging on page 173. |
| `NSPlumbing` | NameServer | Turns on logging for the NameServer component.<br>For log entry details, see NameServer logging on page 151. |
| `ProEvents.UI.Char`<br>`ProEvents.UI.Command`<br>`ProEvents.Other` | ABL clients, AppServer, and WebSpeed agents | Turns on event logging for different categories of events.<br>For log entry details, see Event logging on page 139. |

| Log entry type | Executable | Description |
|---|---|---|
| QryInfo | ABL clients, AppServer, and WebSpeed agents | Turns on logging of queries (each open query and FOR EACH block) executed in an application.<br>For log entry details, see Query information logging on page 152. |
| SAX | ABL clients, AppServer, and WebSpeed agents | Turns on logging for the SAX parser.<br>For log entry details, see SAX parser logging on page 165. |
| Temp-Tables | ABL clients, AppServer, and WebSpeed agents | Enables specific logging for temp-tables so that application developers and technical support can gather more information about the temp-tables used by an application.<br>For log entry details, see Temp-table logging on page 166. |

| Log entry type | Executable | Description |
|---|---|---|
| `TTStats` | ABL clients, AppServer, and WebSpeed agents | Enables logging of temp-table statistics to the corresponding server or log files.<br>For more information on TTStats, see Collecting temp-table usage statistics on page 181. |
| `UBrokerBasic`<br>`UBroker.ClientFSM`<br>`UBroker.ServerFSM`<br>`UBroker.ClientMsgStream`<br>`UBroker.ServerMsgStream`<br>`UBroker.ClientMsgQueue`<br>`UBroker.ServerMsgQueue`<br>`UBroker.ClientMemTrace`<br>`UBroker.ServerMemTrace`<br>`UBroker.ThreadPool`<br>`UBroker.Stats`<br>`UBroker.AutoTrim`<br>`UBroker.All` | Unified Broker | Turns on logging for the Unified Broker component.<br>For log entry details, see Temp-table logging on page 166. |
| `WSADefault`<br>`4GLProvider`<br>`Actional`<br>`AdminProvider`<br>`BrokerClient`<br>`DynamicApi`<br>`MsgDebug`<br>`PoolMgmt`<br>`Properties`<br>`RefCounts`<br>`RunProcs`<br>`SessionPool`<br>`SOAPProc`<br>`WSA`<br>`WSAObject`<br>`WSAObjectPool`<br>`WSDLDoc` | Web Services Adapter (WSA) | Turns on logging for the Web Services Adapter component.<br>For log entry details, see Web Services Adapter (WSA) and Web Service logging on page 168. |
| `DS.Performance` | DataServer clients | Turns on logging for the client-side performance details.<br>For log entry details, see DataServer client context logging on page 128. |

## Log entry types (Server context)

The following table lists the available log entry types in the server context.

**Table 33: Log Entry Types (Server context)**

| Log Entry Type | Executable | Description |
|---|---|---|
| SQL | DataServer broker | Turns on logging of SQL query processing in the DataServer log file. |
| Cursor | DataServer broker | Turns on logging of cursor details in the DataServer log file. |
| Trans | DataServer broker | Turns on logging for the processing of transactions and subtransactions in the DataServer log file. |
| Connects | DataServer broker | Turns on logging for connection attributes and related information in the DataServer log file. |
| Performance | DataServer broker | Turns on logging of performance related information in the server context. |

For log entry details, see DataServer server context logging on page 129. For a detailed description of the information written to the log file for each of these log entry types, see Log Entry Types Detail on page 121.

## Specifying log entry types

You can specify one or more log entry types to write to the log file by using:

- The Log Entry Types (`-logentrytypes`) startup parameter
- The `LOG-ENTRY-TYPES` attribute on the `LOG-MANAGER` system handle
- The `logEntryTypes` and related properties in the `ubroker.properties` file

The following table shows how to specify log entry types.

**Table 34: Specifying log entry types**

| Method | General syntax | Example |
|---|---|---|
| Startup parameter | `-logentrytypes` *log-entry-type* [ `:` *level* ] | `-logentrytypes 4GLTrace:2,DynObjects.*:3` |
| `LOG-MANAGER` system handle | `LOG-MANAGER:LOG-ENTRY-TYPES = "` *log-entry-type* [ `:` *level* ] `"` | `LOG-MANAGER:LOG-ENTRY-TYPES = "DB.Connects,4GLTrace:2,DynObjects.UI:3"` |
| OpenEdge Management or OpenEdge Explorer, or `ubroker.properties` files | `logEntryTypes = "` *log-entry-type* [ `:` *level* ] `"srvrLogEntryTypes = "` *log-entry-type* [ `:` *level* ] `"brkrLogEntryTypes = "` *log-entry-type* [ `:` *level* ] `"` | `logEntryTypes = "DB.Connects,4GLTrace:2,DynObjects.UI:3"` |
| `DSLOG-MANAGER` system handle | `DSLOG-MANAGER = "log-entry-type [ :level ] "` | `DSLOG-MANAGER:LOG-ENTRY-TYPES = "Cursor,DS.Interface:3"` |

For a detailed description of the Log Entry Types (`-logentrytypes`) startup parameter, see *Startup Command and Parameter Reference*.

For a detailed description of the `LOG-MANAGER` system handle, see *ABL Reference*.

For more information on `ubroker.properties`, see the `ubroker.properties.README` file in `$INSTALLDIR/properties/ubroker.properties.README`.

To use OpenEdge Management or OpenEdge Explorer to view and set properties, see Logging settings on page 119.

For information about specifying log entry types for the server context of DataServers, see *Use the Microsoft SQL Data Server*, and *Use the Oracle Data Server*.

# Logging levels

Logging levels represent the amount of logging information written to the log file. The following table lists the valid logging levels.

**Table 35: Logging levels**

| Logging level | Description |
|---|---|
| 0 (None) | Logs no entries. This is equivalent to turning logging off. |
| 1 (Errors) | Logs OpenEdge error messages. This includes all error messages and is unrelated to the entry types specified. Errors continue to be logged at all higher levels. |

| Logging level | Description |
|---|---|
| 2 (Basic) | Log entry type determines the logged information. Each log entry type generates at least some output. This is the default. |
| 3 (Verbose) | Log entry type determines the logged information. Log messages include the information logged at levels 1 and 2. |
| 4 (Extended) | Log entry type determines the logged information. |

By default, the logging level you specify applies to all log entry types. However, you can specify a different logging level for individual log entry types. The higher logging levels include the log information about the lower logging levels, in addition to the information logged at that level. For more information, see Specifying log entry types on page 116.

**Note:** To turn off logging without modifying other logging characteristics, set the logging level to 0 (None). If you do not specify a log file, but do specify other logging parameters, you receive a warning message that alerts you to this inconsistency.

## Specifying the logging level

You can specify the level at which log entries are written to the log file by setting:

- The Logging Level (`-logginglevel`) startup parameter
- The `LOGGING-LEVEL` attribute on the `LOG-MANAGER` system handle
- The `loggingLevel` and related properties in the `ubroker.properties` file

The following table shows how to specify logging levels.

**Table 36: Specifying logging levels**

| Method | General syntax | Example |
|---|---|---|
| Startup parameter | `-logginglevel`*`level`* | `-logginglevel 2` |
| `LOG-MANAGER` system handle | `LOG-MANAGER:LOGGING-LEVEL = `*`level`* | `LOG-MANAGER:LOGGING-LEVEL = 2` |
| OpenEdge Management or OpenEdge Explorer, or `ubroker.properties` files | loggingLevel = *`level`*`srvrLoggingLevel = `*`level`*`brkrLoggingLevel = `*`level`* | `logginglevel = 2` |
| `DSLOG-MANAGER` system handle | `DSLOG-MANAGER:LOG-ENTRY-TYPES "log-entry-type [:level] "` | `DSLOG-MANAGER:LOG-ENTRY-TYPES = "Cursor,DS.Interface:3"` |

For a detailed description of the Logging Level (`-logginglevel`) startup parameter, see *Startup Command and Parameter Reference*.

For a detailed description of the `LOG-MANAGER` system handle, see *ABL Reference*.

For more information on `ubroker.properties`, see the `ubroker.properties.README` file in `$INSTALLDIR/properties/ubroker.properties.README`.

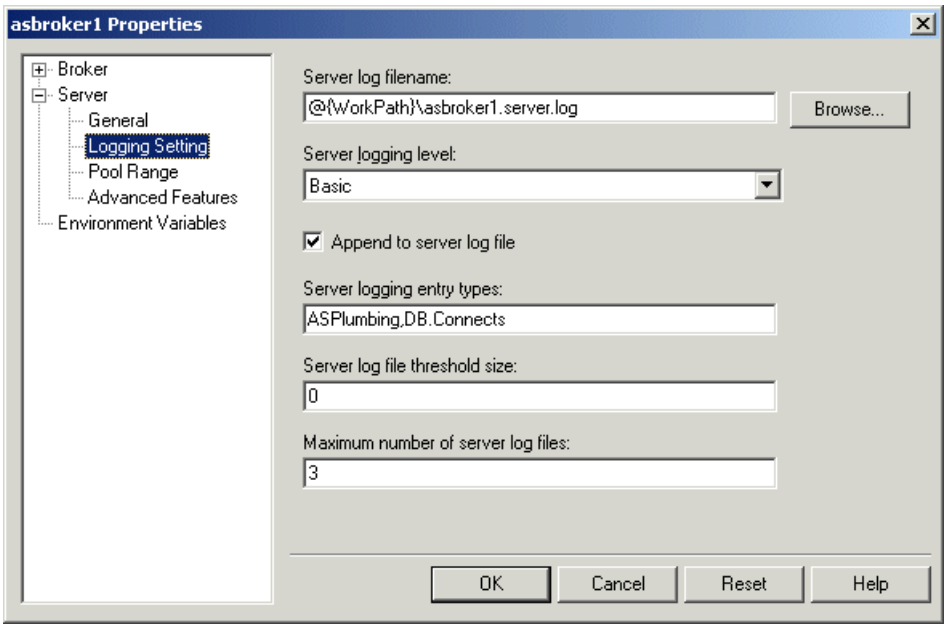
To use OpenEdge Management or OpenEdge Explorer to view and set properties, see Logging settings on page 119.

For information about specifying logging level for the server context of DataServers, see *Use the Microsoft SQL Data Server*, and *Use the Oracle Data Server*.

# Logging settings

To access the AppServer properties (for example):

1. Open OpenEdge Management or OpenEdge Explorer.
2. Connect to **localhost**.
3. Expand AppServer.
4. Right-click **asbroker1** and choose **Properties** from the pop-up menu.
5. Expand **Server** and choose **Logging Setting**. The following dialog box appears:




This view shows the defaults for the various logging settings.

**Note:** You can also use the `mergeprop` utility installed with OpenEdge to manually edit the `ubroker.properties` file. For information on using `mergeprop`, see *Configure OpenEdge*.

Log settings in the OpenEdge Management and OpenEdge Explorer also pertain to the client context for DataServers. However, a different set of properties in the ubroker.properties file apply to logging for DataServer broker instances. To acquire information about log settings in the ubroker.properties file for DataServer brokers, see *Use the Microsoft SQL Data Server*.

# 6

# Log Entry Types Detail

This chapter provides a detailed description of the information written to the log file for each of the log entry types.

For details, see the following topics:

- ABL message logging
- ABL trace logging
- ABL transaction logging
- AIA logging
- AppServer logging
- Database connection logging
- DataServer logging
- Dynamic object logging
- Event logging
- File access logging
- Ignored operation logging
- NameServer logging
- Query information logging
- SAX parser logging

- Temp-table logging
- Unified Broker logging
- Web Services Adapter (WSA) and Web Service logging
- WebSpeed Messenger logging

# ABL message logging

Specifying the ABL message (`4GLMessages`) log entry type turns on logging of ABL (Advanced Business Language) messages. It causes OpenEdge to write all ABL `VIEW-AS ALERT-BOX` messages to the log file. If the Debug Alert (`-debugalert`) startup parameter is used, the ABL stack is also written to the log file at the time of the message. Information is logged at Level 2 (Basic) and higher. Log entries generated for this type contain the tag `4GLMESSAGE`.

**Note:** You do not need to turn on ABL message logging to have messages logged for the AppServer or WebSpeed.

# ABL trace logging

Specifying the ABL trace (`4GLTrace`) log entry type turns on logging for the execution of internal procedures, user-defined functions, persistent triggers, named events, and class events (generated by the `RUN`, `FUNCTION`, `PUBLISH`, and `SUBSCRIBE` statements, and (for class events) by the `Publish( )`, `Subscribe( )`, and `Unsubscribe( )` event methods, respectively). Class events can also include subscribing and unsubscribing handlers for .NET events and publishing ABL implementations of inherited .NET abstract events. It also logs the instantiation and use of classes, including execution of constructors (invoked by the `NEW` phrase and also by the `SUPER` and `THIS-OBJECT` statements), the execution of methods defined within classes (including those invoked using the `SUPER` system reference), the execution of property accessor methods (invoked by accessing a property of a class), and the execution of destructors (invoked by executing the `DELETE OBJECT` statement).

ABL trace logging is available on all ABL clients including GUI clients, character clients, WebSpeed agents, and AppServer agents. Log entries generated for this type contain the tag `4GLTrace`.

ABL trace logging logs an *entry message* when execution begins. Entry messages are logged at Level 2 (Basic) and higher. ABL trace logging also logs a *return message* to indicate when the application returned. Return messages are logged at Level 3 (Verbose) and higher.

## Syntax

The general format for ABL trace log messages is:

```
type routine-name  [PERS TRIG] [in routine-name] [PERSIST] "input-param" [caller
 - source @ line-number]
```

*type*

The log message entry type. The following table lists the log message entry types.

**Table 37: ABL trace log message entry types**

| Type | Description |
|---|---|
| Delete | Deleting class instances |
| Function | Invoking user-defined functions |
| Invoke | Invoking methods |
| New | Instantiating classes |
| Publish | Publishing both named and class events |
| Run | Running procedures |
| Subscribe | Subscribing handlers for both named and class events |
| Super | Invoking super class methods |
| Unsubscribe | Unsubscribing handlers for class events only |

*routine-name*

Procedure or function name being executed. For classes, this is the name of the class being instantiated or destroyed, or the name of the invoked method.

PERS TRIG

Indicates that the procedure runs as a result of a persistent trigger. In this case, this is the only piece of information in the log entry after the *routine-name*.

in *routine-name*

For procedures, logs the calling *routine-name*. For example, RUN*xx*in*yy* statement, logs *yy* as the *routine-name*. For functions and methods, logs the function or method when the function or method is not located in the caller procedure or class.

PERSIST

Indicates the RUN statement has the PERSISTENT clause.

*input-param*

List of input and input-output parameters, each separated by a single space.

*caller*

Indicates where the call to the statement or function is: Main block, SYSTEM-TRIGGER, or the name of an internal procedure, function, or method.

*source*

Indicates the name of the procedure or class containing the caller.

*@ line-number*

Indicates the line number at which the procedure *routine-name* is executed by the caller routine. When implicitly invoking the constructor of the super class, this is the string IMPLICIT.

For example:

```
4GLTrace Run myproc in myproc2.p "my parameters" [init - foo.p @ 10]
```

The following example shows ABL trace logging for classes:

```
4GLTrace New temp [Main Block - foo.p @ 10]
4GLTrace Super tempbase [temp - temp @ 5]
4GLTrace Invoke bar in temp "3" [Main Block - foo.p @ 11]
4GLTrace Delete temp [Main Block - foo.p @12]
```

The general format for ABL trace log return messages is:

### Syntax

```
Return from routine-name "output-param" [program-name] [STOP | ERROR]
```

*routine-name*

Name of internal procedure, function, method, user-interface trigger, PUBLISH statement, or Main Block.

*output-param*

List of OUTPUT, INPUT-OUTPUT, and RETURN parameters.

*program-name*

Name of the program containing the called procedure.

For example:

```
4GLTrace Return from myproc "my parameters" [myproc2.p]
```

The return message includes the STOP or ERROR tag in cases where one of those conditions is raised during execution.

## ABL transaction logging

Specifying the ABL transaction (4GLTrans) log entry type turns on logging of transactions and subtransactions in ABL procedures. ABL transaction logging logs a message whenever a transaction or subtransaction begins or ends, or when a transaction is undone. Logging ABL transaction activity is useful in identifying transaction processing or scoping errors in your application.

The 4GLTrans log entry type provides information at the following logging levels:

- **Level 2 (Basic)** — Logs messages when a transaction begins or ends, or when a transaction is undone
- **Level 3 (Verbose)** — Logs messages when a transaction or subtransaction begins or ends, or when a transaction or subtransaction is undone

The format for ABL transaction log messages is:

## Syntax

```
{ BEGIN | END | UNDO } [ SUB- ] TRANS trans-id [ proc-name @ line-number ]
```

BEGIN

Indicates the beginning of a transaction or subtransaction block.

END

Indicates the end of a transaction or subtransaction block, where any changes made during the transaction or subtransaction were committed.

UNDO

Indicates the end of a transaction or subtransaction block, where any changes made during the transaction or subtransaction were undone (rolled back).

SUB-TRANS

Indicates the log entry is for a subtransaction (occurring within the scope of the currently active transaction).

TRANS

Indicates the log entry is for a transaction.

*trans-id*

A unique transaction identifier, which lets you locate the matching BEGIN and END messages for a single transaction or subtransaction block within a log file.

*proc-name*

The name of the procedure in which the transaction or subtransaction occurred.

@ *line-number*

The line number (in the debug listing for the procedure specified in *proc-name)* at which the transaction began or ended.

The following procedure and log file output excerpts demonstrate transaction logging:

```
/* trantest.p */
DEFINE VARIABLE iVar AS INTEGER NO-UNDO.
DEFINE VARIABLE i1   AS INTEGER NO-UNDO.
DEFINE VARIABLE i2   AS INTEGER NO-UNDO.
DO TRANSACTION ON STOP UNDO, LEAVE:
  iVar = 1.
  DO TRANSACTION ON STOP UNDO, LEAVE:
    DO i1 = 1 TO 10:
      i2 = i2 + i1.
    END.
    iVar = 2.
    STOP.
  END.
END.
MESSAGE ivar VIEW-AS ALERT-BOX INFO BUTTONS OK.
QUIT.
```

The preceding procedure generates the following log file output for transaction logging at level 2 (Basic):

```
BEGIN TRANS 702 [trantest.p @ 6]
END TRANS 702 [trantest.p @ 15]
```

And the following log file output for transaction logging at level 3 (Verbose):

```
BEGIN TRANS 702 [trantest.p @ 6]
BEGIN SUB-TRANS 703 [trantest.p @ 8]
UNDO SUB-TRANS 703 [trantest.p @ 14]
END TRANS 702 [trantest.p @ 15]
```

For more information about the processing of transactions and subtransactions, see the chapter on transactions in *Develop ABL Applications*.

# AIA logging

Specifying an AIA log entry type turns on logging for the AIA component. Following are the AIA log entry types and the information they provide based on logging level:

- **`AiaMgmt`** — Logs messages about regular AIA operations, such as AIA version information, instantiation, some other informational messages, and information on the AIA connection pool. Level 2 (Basic) logs instantiation and informational messages. Level 3 (Verbose) logs the connection pool information. Level 4 (Extended) logs the same information as level 3.
- **`AiaProp`** — Logs the properties set for the specific AIA instance. These properties are set in the `ubroker.properties` file. Information is logged at logging level 2 (Basic) and higher.
- **`AiaRqst`** — Logs information on the `GET` and `POST` commands sent by the client and the response back to the client. Information is logged at logging level 2 (Basic) and higher.
- **`AiaUBroker`** — Logs information on the communication between the AIA and the Unified broker (and AppServer or OpenEdge Adapter for SonicMQ BrokerConnect). Logging level 2 (Basic) provides most information. Level 3 (Verbose) also logs a binary dump of messages exchanged between the client and the broker/server process.

- **`AiaDefault`** — Combines `AiaMgmt` and `AiaProp`. This is the default entry type for the AIA component.

The following example shows entries for the AIA entry type with the logging level set to VERBOSE (3):

```
2 AIA AiaMgmt     AIA v100A (25-Apr-03)
2 AIA AiaProp     installDir                : d:/dlc100a
2 AIA AiaProp     properties file           : d:/dlc100a/properties/ubroker.properties
2 AIA AiaProp     instanceName              : Aia1
2 AIA AiaProp     logfilename               : d:\progress\wrk100a\Aia1.aia.log
2 AIA AiaMgmt     instanceName: Aia1
2 AIA AiaMgmt     InstallDir: d:/dlc100a
2 AIA AiaMgmt     propertyFileName: d:/dlc100a/properties/ubroker.properties
2 AIA AiaMgmt     AIA_HTTPS is licensed in this configuration.
2 AIA AiaRqst     Start: doGet()
2 AIA AiaRqst     Requested URL: http://pcspot/aia/Aia
2 AIA AiaRqst     Request information:
2 AIA AiaRqst     Request method : HEAD
3 AIA AiaUBroker  readMsg()
3 AIA AiaUBroker  mark() : markpos=0
3 AIA AiaUBroker  readubhdr
3 AIA AiaUBroker  appUserid=
3 AIA AiaUBroker  arbString=
3 AIA AiaUBroker  appService= asbroker1
2 AIA AiaUBroker  processConnect() urlhttp://localhost:5162/asbroker1
3 AIA AiaUBroker  [SC-000001] nameserverLookup() : brokerhost= 172.16.108.180 brokerport=
 3090
3 AIA AiaUBroker  [SC-000001] wrote UBRQ_CONNECT to (172.16.108.180:3090)
```

# AppServer logging

Specifying an AppServer log entry type turns on logging for the AppServer.

The ASPlumbing log entry type provides information at the following logging levels:

- **Level 2 (Basic)** — Logs messages when an AppServer agent starts and stops, and when client processes connect and disconnect from the AppServer agent.
- **Level 3 (Verbose)** — Same as logging level 2, plus logging of the execution of the top level procedure (that is, the procedure that the client process asks the AppServer agent to execute on its behalf, but not any procedure the top level procedure executes).
- **Level 4 (Extended)** — Same as logging level 3, plus messages about server state, and status messages about read and write socket operations. Do not turn this on unless directed to do so by Progress Software Corporation technical support.

Log entries generated for this type contain the tag `AS`. An example of AppServer logging messages (with headers suppressed) is shown:

```
AS Starting application server for asbroker1. (5560)
AS Application Server Startup. (5473)
AS Shutdown request received. (5465)
AS Application Server Shutdown. (5476)
```

The ASDefault log entry type combines the `ASPlumbing` and `DB.Connects` log entry types. It is the default value for AppServer agents.

# Database connection logging

Specifying the database connection (`DB.Connects`) log entry type turns on logging of database connections (connects and disconnects). The messages include database name and user ID number. Information is logged at logging level 2 (Basic) and higher.

Log entries generated for this type contain the tag `CONN`. An example of database connection logging messages (with headers suppressed) is shown:

```
CONN Connected to database sports, single user. (9542)
CONN Disconnecting from database sports, single user. (9544)
```

The default for AppServer is `ASPlumbing` and `DB.Connects`. Generally, these log entry types are together in an AppServer log. For example:

```
AS Starting application server for asbroker1. (5560)
AS Application Server Startup. (5473)
CONN Connected to database sports, single user. (9542)
AS Shutdown request received. (5465)
AS Application Server Shutdown. (5476)
CONN Disconnecting from database sports, single user. (9544)
```

# DataServer logging

Specifying DataServer log entry types turns on logging for the DataServer client and the server contexts. The log entry types do not control the generic logging of runtime errors and messages during execution. The logging level specified with the log entry type controls the amount of information to be logged. The logging levels range from 0 to 4, although specific log entry types use a logging level value between 2 and 4.

**Note:** The logging levels range from 0 to 4, both for the client and server contexts. Level 0 logs no entries and using that level is equivalent to turning the logging off. Level 1 logs all error messages and is unrelated to the entry type specified. Errors continue to be logged at higher logging levels. Level 4 is reserved for future releases and no additional logs are generated for most of the log entry types.

## DataServer client context logging

The `DS.QryInfo` log entry type provides information at the following logging levels:

- **Level 2 (Basic)** — Logs basic information about client generation of SQL query generation. Information about find mode, field list declarations, selected index, BY clauses, file type, join level and lock mode are logged.
- **Level 3 (Verbose)** — Logs details about SQL queries in addition to those generated using level 2. Information about joins, client selection derivations from SMLOCAL, SMGLOBAL and SMJOINALIEN ditem designations, prep time clause variations and the compiler time clause variations, and messages regarding field substitutions are logged.

**Note:** This client entry type is closely associated with a server context entry type known as "SQL". See *Use the Microsoft SQL Data Server*, and *Use the Oracle Data Server* for more details on log entry types for the server context.

The `DS.Cursor` log entry type logs client side cursor details at the following logging levels:

- **Level 2 (Basic)** — Logs messages pertaining to cursor details such as table-number and cursor ID. Cursor ID is also tracked to the server so it can be tracked in the server context.
- **Level 3 (Verbose)** — Logs information on findmode, lockmode, file type, join level, selected index, scrolling flags, lookahead settings, bind mode, client selection, in addition to the messages logged in level 2 of the client context.

**Note:** This client entry type is closely associated with a server context entry type known as "Cursor". See the DataServer for MS SQL Server Guide for more details on log entry types for the server context.

The `4GLTrans` log entry type is not specific to DataServers but its output may be mildly useful as a supplement in combination with the output of a server context entry type known as `TRANS`.

The `4GLTrans` log entry type provides information specific to DataServers at the following logging levels:

- **Level 2 (Basic)** — Logs messages when a transaction begins or ends. The incomplete transactions are also logged at this level.
- **Level 3 (Verbose)** — Logs messages when either a transaction or a sub-transaction begins or ends. Also, the incomplete transactions and sub-transactions are logged at this level.
- **Level 4 (Extended)** — Logs client lock mode, transaction isolation level and other transaction-related information as pertains to DataServers in the client context. It lists when records are re-tested for server modification, when foreign connect environments are cleared, when locks are released, and when transactions are aborted.

The `DB.Connects` log entry type provides the client-side connection related information at the following logging levels:

- **Level 2 (Basic)** — Logs basic client-side information such as the database name and user name. It also logs the database ID, appends logical database alias, and the connect string information.
- **Level 3 (Verbose)** — Logs information about the `-Dsrv` switches used during the connection time, along with information on the ports and the IP address, and Protocol version.

The `DS.Performance` logs the performance related information at the following logging levels:

- **Level 2 (Basic)** — Logs the time taken for calls to the federated-layer entry point functions and the time spent at the DataServer client layer. This reports the session level time spent at client layer.
- **Level 3 (Verbose)** — Logs the relative time spent on the function call to that spent on the DataServer Client Layer. This reports the connection level time spent at client layer.
- **Level 4 (Extended)** — Logs time spent for the "individual roundtrip" at the client layer.

## DataServer server context logging

The `SQL` log entry type provides information on the preparation and execution of SQL queries on the MS SQL and Oracle databases in the server context at the following levels:

**Note:** ABL procedure and Line Numbers are logged using the `-Dsrv "PRGRS_LOG_ABL"` connection switch with the `SQL` log entry type at any level.

- **Level 2 (Basic)** — Logs some server log information related to the legacy `qt_debug,SQL` option, and SQL query preparation.
- **Level 3 (Verbose)** Logs the bind values if SQL uses Bind parameters and reports the "SQL CRC".

The `Cursor` log entry type is mapped to the legacy `qt_debug,CURSOR` option and provides information about use of cursors in the server context at the following levels:

- **Level 2 (Basic)** — Logs all existing cursor messages mapped from the legacy `qt_debug` option and the new and basic information on cursor such as table-number and client-cursor ID or server-cursor ID.
- Level 3 (Verbose) — Logs information on cursor status, reposition of cursor, index used, sort flags, usage of cursors (block, lookahead, fast, slow), cache type, fetch source/target, fetched records, and average buffer length.

The `Trans` log entry type provides server-side transaction information at the following levels:

- **Level 2 (Basic)** — Logs messages when an SQL transaction begins or ends on the server. Logs information about the undone transactions on the MS SQL and Oracle databases.
- **Level 3 (Verbose)** — Logs messages when a transaction (including the transaction level) begins or ends. Logs information about the undone transactions and sub-transactions.
- **Level 4 (Extended)** — Logs messages about records locked in a transaction including the type of lock and lock upgrades.

The `Connects` log entry type provides the server-side connection related information at the following levels:

- **Level 2 (Basic)** — Logs information pertaining to the `-Dsrv` switches such as lockwait timeout, transaction isolation level, etc. and the set parameters used on the connection. Logs all DataServer connection attributes established before the start and after the end of a session.
- **Level 3 (Verbose)** — Logs reports on the additional connection attributes.

The `Performance` log entry type provides the performance related information in the server context at the following levels:

- **Level 2 (Basic)** — Logs the time of call to the server-layer entry point functions. Logs the time spent on the DataServer server layer. This reports the session level time spent at the server layer.
- **Level 3 (Verbose)** — Logs the relative time spent on the function call to that spent on the Dataserver server layer. This reports the session level time spent at the server layer.
- **Level 4 (Extended)** — Logs time spent for the "individual roundtrip" at the server layer.

## DataServer logging switch for ABL to SQL correlation

The `PRGRS_LOG_ABL` connection switch enables logging of the ABL to SQL correlation on the MS SQL and Oracle DataServers. The DataServer log file provides information, to the DataServer, about every ABL statement that executes on the foreign data sources and the correlated SQL query statements.

You use the `SQL` log entry type only to obtain log output that consists of information on SQL query in the server context. However, with the `PRGRS_LOG_ABL` switch, you can obtain the ABL code information for the entire server context, independent of the specific log entry types selected on the server. You can build a correlative intelligence into the functional domain of any other log entry type on the server.

Like the other Enhanced Logger log entry types, even the PRGRS_LOG_ABL connection switch is not turned on by default. If you want to log information for the component domain of any specific log entry type, you select that log entry type either at startup or during runtime of an application. In the server context, you can use the -Dsrv connection switch to define a log entry type at startup.

Similarly, you can set the client context switch, PRGRS_LOG_ABL, using the -Dsrv switch at client startup using the following syntax:

```
-Dsrv logentrytypes,SQL,PRGRS_LOG_ABL,n
```

where, n indicates a switch value between 0 and 2 as listed in the following table.

**Table 38: PRGRS_LOG_ABL connection switch values**

| Switch Value | Description |
|---|---|
| 0 | Disables the ABL to SQL correlation feature. No correlation information will be logged. |
| 1 | Enables the ABL to SQL correlation feature for SQL-related log entry types. Every SQL request derived from the log entry type output is prefixed with ABL filename and line number which can be correlated to a compiled listings file. |
| 2 | Enables a superset of value 1 which, along with correlation information, includes visual delimiters marking the start and end of each ABL file module. |

**Note:** The logging levels range from 0 to 4, although specific log entry types use a logging level value between 2 and 4. The logging level specified with the log entry type controls the degree of detail to be logged. The information obtained covers various attributes of the DataServer both in the client and server contexts. The PRGRS_LOG_ABL switch enables only the logging of ABL to SQL correlation.

For more information on DataServer logging, see *Use the Microsoft SQL Data Server*, and *Use the Oracle Data Server*.

# Dynamic object logging

Specifying a dynamic object log entry type turns on logging of dynamic object creation and deletion. The following table lists the dynamic object log entry types and their associated objects.

**Table 39: Dynamic objects that are logged**

| Log entry type | Type of objects logged | Objects logged |
|---|---|---|
| DynObjects.Class | Classes | Logs the creation and deletion of classes for:<br>• Progress.Lang.Object |

| Log entry type | Type of objects logged | Objects logged |
|---|---|---|
| DynObjects.DB | Database-related objects | • BUFFER<br>• DATASET<br>• DATA-SOURCE<br>• QUERY<br>• TEMP-TABLE<br><br>**Note:** For BUFFERs, QUERYs, and TEMP-TABLEs, does not include handles for static objects. For BUFFER objects, the log file includes the table name associated with the buffer. |
| DynObjects.Other | Objects that do not fall into the other groups | • ASYNC-REQUEST<br>• CALL<br>• CONTROL-FRAME<br>• MEMPTR<br>• PROCEDURE<br>• SERVER<br>• SERVER-SOCKET<br>• SOCKET<br><br>**Note:** For ASYNC-REQUEST, the log entry includes the procedure name in the RUN statement. For MEMPTR variables, logs an entry each time memory is allocated or freed. The log entry includes the allocation size (in bytes), and the handle ID always is 0. For PROCEDUREs, does **not** includes handles for non-persistent procedures (for example, THIS-PROCEDURE). The log entry includes the name of the persistent procedure. |

| Log entry type | Type of objects logged | Objects logged |
|---|---|---|
| DynObjects.UI | User-interface- related objects | • COMBO-BOX<br>• BROWSE<br>• BROWSE-COLUMN<br>• BUTTON<br>• EDITOR<br>• FILL-IN<br>• FRAME<br>• IMAGE<br>• MENU<br>• MENU-ITEM<br>• RADIO-SET<br>• RECTANGLE<br>• SELECTION-LIST<br>• SLIDER<br>• SUB-MENU<br>• TEXT<br>• TOGGLE-BOX<br>• WINDOW<br><br>**Note:** For all objects, the log entry includes the object name, when available. For BROWSE-COLUMNs, the log entry does not include handles associated with static browse objects. If there is no value for the Name attribute (for calculated fields, for example), the log entry contains the Label attribute. Also, includes the BROWSE name, or the BROWSE handle, if there is no name. |
| DynObjects.XML | XML-related objects | • SAX-ATTRIBUTES<br>• SAX-READER<br>• SAX-WRITER<br>• SOAP-HEADER<br>• SOAP-HEADER-ENTRYREF<br>• X-DOCUMENT |

| Log entry type | Type of objects logged | Objects logged |
|---|---|---|
| | | • X-NODEREF |

## Logging dynamic object creation and deletion

To create log entries for dynamic object creation and deletion, use the Log Entry Types (`-logentrytypes`) startup parameter or, for AppServer and WebSpeed, the s`rvrLogEntryTypes` property in the `ubroker.properties` file. Specify one or more of the four dynamic object log entry types in a comma-separated list, or specify `DynObjects.*` to get all four types. For example:

```
-logentrytypes DynObjects.*
```

```
-logentrytypes DynObjects.DB,DynObjects.Other
```

You can also turn on this logging at run time, by using the `LOG-MANAGER` system handle. For example, you can include the following line in the ABL code:

```
LOG-MANAGER:LOG-ENTRY-TYPES = "DynObjects.UI"
```

You can turn off this logging in the ABL application by resetting the attribute. For example:

```
LOG-MANAGER:LOG-ENTRY-TYPES = ?
```

Log entries generated for any of these types contain the tag `DYNOBJECTS`.

OpenEdge writes a log entry every time it creates or destroys a dynamic object tracked by this feature. This also includes allocation and deletion of space for `MEMPTR`s, if you specify `DynObjects.Other` or `DynObjects.*`.

### Logging class creation and deletion

Log entries generated for this type contain the tag `DYNOBJECTS`. An example of Dynamic object logging messages for classes (with headers suppressed) is shown:

```
DYNOBJECTS Created        Progress.Lang.Object    Handle:1093 (foo.p Line 18)
    tempclass
DYNOBJECTS Delete Pending Progress.Lang.Object    Handle:1094 (method1     tempclass
Line 13) tempclass
DYNOBJECTS Deleted        Progress.Lang.Object    Handle:1093 (foo.p Line 20)
    tempclass
```

## Deleted objects

There are cases where the application deletes an object with a DELETE OBJECT statement, but the AVM delays the deletion of the object. One example is an output TABLE-HANDLE parameter. If the ABL application deletes this object before returning, the AVM delays the deletion so it can transfer the output temp-table records back to the calling program when the program returns. If the AVM delays the object deletion, a log entry indicates that the deletion is pending. A log entry also indicates when the AVM deletes the object. This can happen with the following object types:

- BUFFER (associated with an output TABLE-HANDLE parameter)
- DATASET (for example, an OUTPUT DATASET-HANDLE parameter)
- PROCEDURE (for example, if a procedure contains DELETE THIS-PROCEDURE)
- TEMP-TABLE (for example, an OUTPUT TABLE-HANDLE parameter)

## Objects created and deleted implicitly

The AVM creates and deletes some objects implicitly. This means the ABL application does not explicitly have a CREATE or DELETE OBJECT statement for the object. An example of implicit object creation is a SOCKET object, which the AVM creates when it receives a Connect event on a SERVER-SOCKET. In this case, the log entry contains the word IMPLICIT. The AVM creates the following object types implicitly:

- ASYNC-REQUEST
- BUFFER (that is, the default buffer of a dynamic query)
- SOAP-HEADER (which can also be created explicitly)
- SOAP-HEADER-ENTRYREF
- SOCKET
- TEMP-TABLE (an INPUT or OUTPUT TABLE-HANDLE parameter)
- X-DOCUMENT

An example of implicit object deletion is when the AVM destroys an object during the course of destroying its object pool. In this case, the log entry contains the word POOLDEL.

## Logging levels for dynamic objects

There are two logging levels supported for DynObjects entry types: 2 (Basic) and 3 (Verbose). Setting logging to 4 (Extended), logs information as for 3 (Verbose).

### Basic logging level (2)

The log entries for logging level 2 (Basic) include the following information, and any level above 1 (Errors) logs this information:

- **Action** — For objects, this is Created, Deleted, or Delete Pending. For MEMPTRs, this is Allocated or Deallocated.
- **Object type** — For example, BUFFER, SERVER, or BUTTON.
- **Handle value** — Handle: followed by the INTEGER value of the handle (0 for MEMPTRs).

- **Routine name** — The name of the `.p` or `.w` file or method where this application created or destroyed the object.
- **Line number** — The line number in the `.p` or `.w` file or method where the application created or destroyed the object. This line number comes from the debugger listing file. When the AVM creates a `SOCKET` object implicitly, the line number always is `0`.

  For other implicitly created objects, the line number is meaningful. For example, for an `ASYNC-REQUEST` handle, it is the line number of the `RUN` statement.
- **`IMPLICIT`** — Appears when the AVM created the object implicitly.
- **`POOLDEL`** — Appears when the AVM deleted the object because it was deleting the object's object pool.
- **Other** — For specific object types, the log output line includes other attributes, including:
  - `Name:` followed by the object name. For `BUFFER` objects, the `Name` attribute is the name of the table associated with the buffer, so the entry contains `Table:` followed by the table name.
  - `TEMP-TABLE` objects do not show the `Name` attribute. This information is not available when the application creates the `TEMP-TABLE` object, but only after it invokes the `TEMP-TABLE-PREPARE` method.
  - The allocation's size, for `MEMPTR`s.
  - The procedure name for `PROCEDURE` objects.
  - The name of the procedure associated with an `ASYNC-REQUEST-HANDLE` object (that is, the procedure that ran asynchronously).
  - The `Label` attribute for `BROWSE-COLUMN` objects, if there is no `Name` attribute (for example, for calculated fields). Also, to identify the browse parenting the column, the log entry includes the `BROWSE` name or `BROWSE` handle ID.

Here are sample BASIC log entries (with headers suppressed):

```
[05/10/31@14:07:15.892-0400] P-002702 T-002867 2 4GL DYNOBJECTS
Created        MENU               Handle:1098 (x.p Line 4)
[05/10/31@14:07:15.892-0400] P-002702 T-002867 2 4GL DYNOBJECTS
Deleted        MENU               Handle:1098 (x.p Line 5)
```

```
Created        BROWSE             Handle:1139 (y.p Line 29)
[05/10/31@14:07:17.908-0400] P-002702 T-002867 2 4GL DYNOBJECTS
Created        BROWSE             Handle:1139 (y.p Line 29)
[05/10/31@14:07:17.908-0400] P-002702 T-002867 2 4GL DYNOBJECTS
Created        BROWSE-COLUMN      Handle:1143 (y.p Line 63) Name: Cust-Num [Browse
Handle:1139]
```

```
[05/10/31@14:07:17.908-0400] P-002702 T-002867 2 4GL DYNOBJECTS
Created        BUFFER      Handle:1144 (y.p Line 70) IMPLICIT Table:item
```

```
[05/10/31@14:07:15.892-0400] P-002702 T-002867 2 4GL DYNOBJECTS
Deleted        MENU               Handle:1098 (x.p Line 5) POOLDEL
```

## Verbose logging level (3)

Setting the logging level to 3 (Verbose), also logs object-pool related information.

First, level 3 adds pool information to the log entry every time a dynamic object is created that is being tracked, except PROCEDURE, MEMPTR, and BROWSE-COLUMN objects. PROCEDUREs always are in the Session pool. MEMPTR and LONGCHAR memory is not stored in a pool. The log entry for BROWSE-COLUMN objects contains the browse name or handle ID information in the creation log entry (as described above), so you can refer back to the browse creation entry to find out which pool contains the object.

Second, level 3 logs an extra entry every time an object-pool is created or deleted. This helps identify when the AVM deletes an object in the course of deleting an object pool, as opposed to when the application deletes the object explicitly.

There are two formats for the messages generated at level VERBOSE, depending on the action:

- **Format 1** — OpenEdge logs the following when creating or deleting an object pool:
  - **Action** — This is Creating Pool or Deleting Pool.
  - **Pool name** — This is <unnamed>, <Session Pool>, or the name of a named pool.
  - **PERS** — This indicates the pool is persistent. OpenEdge does not log this for Deleting Pool actions.
  - **Routine name and line number** — The routine name is the name of the .p or .w file or method where the AVM created or destroyed the pool. The line number comes from the debugger listing file.

**Note:** The line number can be the string IMPLICIT for an unnamed widget pool if the pool is created implicitly during the instantiation of a user-defined object.

- **FROM *routine-name* —** *routine-name* is the name of the `.p` or `.w` file or method where the AVM created the object pool. OpenEdge logs this only for the `Delete` pool.

Here are sample log entries (with the headers suppressed):

```
Creating pool  <unnamed> (my.p Line 2)

Deleting pool  mypool  (x.p Line 9) FROM x.p

Deleting pool  mypool2  (x.p Line 10) FROM y.p
```

- **Format 2** — OpenEdge adds pool information to the Basic log entries when creating and deleting dynamic objects. OpenEdge logs this additional information:
  - **Pool name** — This is `<unnamed>`, `<Session Pool>`, or the name of a named pool.
  - **FROM *routinename* —** *routinename* is the name of the `.p` or `.w` file or method that created the object pool. OpenEdge logs this only when the pool is not persistent and not the `Session` Pool.

  Here are some sample log entries (with headers suppressed):

```
[05/10/31@14:07:15.892-0400] P-002702 T-002867 3 4GL DYNOBJECTS Created
MENU              Handle:1098 (x.p Line 4) Pool:<unnamed> FROM x.p
```

```
[05/10/31@14:07:15.892-0400] P-002702 T-002867 3 4GL DYNOBJECTS Created
MENU              Handle:1098 (x.p Line 4) Pool:<unnamed> FROM z.p
```

```
[05/10/31@14:07:15.892-0400] P-002702 T-002867 3 4GL DYNOBJECTS Deleted
MENU              Handle:1098 (x.p Line 5) Pool:mywp FROM x.p
```

```
[05/10/31@14:07:15.892-0400] P-002702 T-002867 3 4GL DYNOBJECTS Deleted
MENU              Handle:1098 (x.p Line 5) Pool:myperswp FROM y.p
```

The following excerpt is from a log file (with headers suppressed) showing an example of how the pool information relates to the other messages:

```
Creating pool  mypool  (x.p Line 2)
Created        MENU             Handle:1098 (x.p Line 4) Pool:mypool FROM x.p
Deleted        MENU             Handle:1098 (x.p Line 5)
Created        TOGGLE-BOX       Handle:1099 (x.p Line 7) Pool:mypool FROM x.p
Deleting pool  mypool  (x.p Line 9) FROM x.p
Deleted        TOGGLE-BOX       Handle:1099 (x.p Line 9) POOLDEL
```

# Event logging

Problems involving event processing present real challenges to ABL developers trying to troubleshoot their code. Event logging takes advantage of the OpenEdge logging infrastructure to track which events occur and the order in which they occur as an ABL program executes. Support personnel can benefit from event logging as they troubleshoot customer code to resolve problems.

## Event generation

An event occurs when the AVM receives notification of an action. An action can be user-driven, such as a mouse click, or some other external event, such as the completion of an AppServer procedure. The AVM also creates events as a result of an APPLY statement, and in response to an external event; for example, pressing the **TAB** key causes LEAVE and ENTRY events to be created.

## Event stack

An external event can result in more than one internal event. For example, clicking the mouse in a fill-in might shift the focus from another field, causing ENTRY and LEAVE events, respectively. Since the ordering of events is significant, the AVM builds a stack based on events as they occur, starting with the first event at the bottom of the stack. Each succeeding event is added to the stack.

The stack is a LIFO (Last In First Out) stack, which means the AVM pushes events onto the stack in the reverse order of their firing. In the focus example above, clicking in the fill-in causes the focus to shift, so the ENTRY event is first in the stack.

In some cases, a trigger on one event takes precedence over a trigger on another event. For example, when you press the mouse button to choose a button, the AVM generates both a MOUSE-SELECT-CLICK event and a CHOOSE event for the button (among other possible events). Since the MOUSE-SELECT-CLICK trigger takes precedence over the CHOOSE trigger for a button, The AVM recognizes that the CHOOSE trigger should not fire and does not push it onto the stack.

Sometimes the AVM has to insert an event elsewhere in the stack than at the top. For example, when you click the mouse to choose a button, the AVM pushes the MOUSE-SELECT-DOWN and ENTRY events that are the first to occur, onto the stack. When you release the mouse button, a CHOOSE event occurs, but if the AVM pushes the CHOOSE event onto the top of the stack, a CHOOSE trigger on the button will fire before an ENTRY trigger. Since an ENTRY trigger always must fire before a CHOOSE trigger, the AVM inserts the CHOOSE event below the ENTRY event so the CHOOSE trigger will fire after the ENTRY trigger.

Once it adds the events for an external action to the stack, the AVM starts to remove events. Since the stack is built in reverse order, the last event pushed onto the stack is the first to be removed. As it removes each event, the AVM checks for triggers and runs the ABL code for each event trigger it finds.

## Logged events

ABL event logging logs the following categories of events:

- **User interface events** — Limited to ABL-owned windows; these events include keyboard activity of both printable and nonprintable keystrokes.
- **ABL-generated events** — For example, ENTRY, LEAVE, and ROW-LEAVE. This also includes Developer Events (U1 through U10).For a complete list of ABL events, see the Events Reference chapter in *ABL Reference*.
- **ASYNC events**

- **Sockets events**
- **COM events**

The following table lists the Windows GUI events that OpenEdge logs. Note, however, that not all GUI events are logged. For example, OpenEdge ignores mouse movements on the basis that they are too numerous and not significant.

**Table 40: Windows GUI events that are logged**

| | |
|---|---|
| WM_WINDOWPOSCHANGING | WM_ACTIVATE |
| WM_DESTROY | WM_CLOSE |
| WM_SETFOCUS | WM_ENABLE |
| WM_KILLFOCUS | WM_LBUTTONUP |
| WM_DROPFILES | WM_LBUTTONDOWN |
| WM_CHAR | WM_LBUTTONDBLCLK |
| WM_KEYDOWN | WM_RBUTTONDOWN |
| WM_SYSKEYDOWN | WM_RBUTTONUP |
| WM_SYSCHAR | WM_RBUTTONDBLCLK |
| WM_MENUCHAR | WM_SYSCOMMAND |
| WM_SIZE | WM_INITMENUPOPUP |

## Logged information

Event logging records information about the following types of actions as they apply to an event:

- **Event occurs** — Either the client receives notification of an external event or the AVM executes an `APPLY` statement.
- **Indirect action taken** — The client takes additional action in response to the event. For example, on a Windows minimize event, the client might minimize several other related windows.
- **ABL event pushed onto stack** — The internal events that result from external events, such as `LEAVE` and `ENTRY`, are pushed onto the stack, from where they will be executed in sequence.
- **Event popped from stack without executing** — In certain cases, the AVM does not execute events on the stack. For example:
  - An event occurs to an object that is disabled. the AVM pushes the event onto the stack, and then pops it off the stack without executing the corresponding trigger, if one exists.
  - Other examples: a trigger executes a `RETURN NO-APPLY` statement; a COM event occurs during a database or socket operation; a field's validation fails.
- **Trigger started** — Begins execution of the user interface trigger associated with the event.

- **Event applied by APPLY statement** — OpenEdge logs each execution of an `APPLY` statement by an ABL program. Frequently, an application calls an `APPLY` statement from within a user-defined trigger.
- **Trigger finished** — Ends execution of the user interface trigger associated with the event.
- **Event popped off stack** — Concludes a specific event execution.

The order of actions presented here is typical of what might appear in a log. Note, however, that not all actions occur for every event. An external event starts all of the actions that occur in an event-driven program, except for initialization. Thus, you would always expect to find the following in the log:

- A GUI, socket, Async, or COM event first
- One or more ABL events and their associated triggers or event procedures
- The conclusion of the external event

The following table lists the information that is logged for each of the described actions. Note that the term *object* refers to the object to which the event applies.

**Table 41: Information logged for each type of action**

| Action | Information logged |
|---|---|
| Event occurs | **Event Type** — `GUI` (GUI Windows), `UI` (character mode), `PRGS`, `COM`, `ASYNC`, or `SOCKET`. **Event Name** — (For example) `WM_xxx`, OCX event name, `LEAVE`, `TAB`.**Widget information** — Type (`BUTTON`, for example), name, and handle ID of the object, when available.**Flag** — Can be set to `TRUE` or `FALSE` for COM events (for use by Progress Software Corporation Technical Support). **OpenEdge Handled** — The string `Progress Handled` appears only for COM object `WM_` events. Logging this string means that the AVM captured the event and will process it, as opposed to passing it to the ActiveX control. |
| Indirect action | **Description** — A descriptive string. When processing an event, the AVM might perform additional tasks that will also be recorded. These extra tasks include: minimize, restore, and close related windows. |
| ABL event pushed onto stack | **Event Name** — (For example) `LEAVE`, `ENTRY`.**Widget information** — Type (`BUTTON`, for example), name, and handle ID of the object, when available.**Procedure Name** — Name of the procedure that created the object.<br><br>**Note:** OpenEdge makes no entry for COM events sent by an ActiveX control. |
| Event popped off stack without executing | **Event Name** — (For example) `LEAVE`.**Event Discarded** — Noted as "discarded." Additional information provided, if the reason for the action is known (for example, "disabled").**Widget information** — Type (`BUTTON`, for example), name, and handle ID of the object, when available. |
| Trigger started | **Handler** — Event name and object name (similar to the `ON` statement), or name of the internal procedure. |

| Action | Information logged |
|---|---|
| Event applied by APPLY statement | **Event Name** — (For example) ENTRY.**Widget information** — Type (BUTTON, for example), name, and handle ID of the object, when available.**Routine and line number** — Name and line number of the routine (.p or .w file or method) and internal procedure, if appropriate, where the APPLY statement is. The line number is based on the Debugger listing file. |
| Trigger finished | **Handler** — Event name and object name (similar to the ON statement), or name of the internal procedure. |
| Event popped off stack | **Event Name** — (For example) LEAVE, ENTRY.**Widget information** — Type (BUTTON, for example), name, and handle ID of the object, when available.**Routine Name** — Name of the routine (.p or .w file or method) that created the object.<br>**Note:** OpenEdge makes no entry for COM events sent by the control. |

## Event log entry types and logging levels

You enable event logging with any of three log entry types. Using three different entry types gives you greater control over what and when information is logged. The following table lists the events that are logged and the logging levels supported for each entry type.

**Table 42: Events logged by log entry type**

| Log entry type | Information logged |
|---|---|
| ProEvents.UI.Char | Turns on logging of keystroke events for printable characters. Logged at level 2 (Basic) and higher. |

| Log entry type | Information logged |
|---|---|
| ProEvents.UI.Command | Turns on logging of keystroke events for nonprintable characters:<br>• **TAB**<br>• **FN** keys<br>• **CTRL**, **SHIFT**, **ALT** and their combinations<br>• **PAGE DOWN**, **PAGE UP**, **HOME**, **END**<br>And Windows GUI events (WM events).Level 2 (Basic) logs nonprintable keystrokes and this subset of WM GUI events:<br>• WM_SETFOCUS<br>• WM_KILLFOCUS<br>• WM_CLOSE<br>• WM_ENABLE<br>• WM_LBUTTONUP<br>• WM_LBUTTONDOWN<br>• WM_RBUTTONUP<br>• WM_RBUTTONDOWN<br>• Developer Events U1–U10<br>Level 3 (Verbose) logs nonprintable keystrokes and all WM GUI events. |
| ProEvents.Other | Turns on logging of COM, Async, and Socket events. Logged at level 2 (Basic) and higher.OpenEdge logs no PUSH and POP entries for COM events. At levels 2 and 3, OpenEdge logs COM events only when the application defines a trigger for the event, on the assumption that you are interested only in events for which you have written a trigger. At level 4 (Extended), OpenEdge logs all COM events. |

## Starting event logging

To create log entries for events, use the Log Entry Types (-logentrytypes) startup parameter or, for the AppServer and WebSpeed, the srvrLogEntryTypes property in the ubroker.properties file. Specify one or more of the three event log entry types in a comma-separated list, or use the * wild-card character to get combinations. For example:

```
-logentrytypes ProEvents.UI.*
```

```
-logentrytypes ProEvents.UI.Command,ProEvents.Other
```

You can also turn on this logging at run time, by using the LOG-MANAGER system handle. For example, you can include the following line in the ABL code:

```
LOG-MANAGER:LOG-ENTRY-TYPES = "ProEvents.UI.Command"
```

You can turn off this logging in the ABL application by resetting the attribute. For example:

```
LOG-MANAGER:LOG-ENTRY-TYPES = ?
```

Log entries generated for any of these types contain the tag ProEvents.

## Event message formats

Event logging message formats correlate closely with event actions. There are six distinct formats.

### Format 1: event occurs

When an event occurs, logged messages contain the following information:

- **Start** — START
- **Identifier** — Event type GUI (GUI Windows), UI (character mode), PRGS, COM, ASYNC, or SOCKET
- **Event name** — For example, LEAVE or GUI WM_*command*. For WM_CHAR, also contains the key pressed
- **Widget information** — Type (BUTTON, for example), name, and handle ID of the object, when available
- **Flag** — T or F (COM events only). For use by Progress Software Corporation Technical Support
- **OpenEdge Handled** — The string Progress Handled appears only for COM object WM_ events

Here are some sample Format 1 log entries:

```
[05/10/31@13:00:00.098-0400] P-000001 T-000002 2 4GL PROEVENTS START PRGS
LEAVE BUTTON Handle:1989
```

```
[05/10/31@12:02:43.570-0400] P-005783 T-005735 2 4GL PROEVENTS  START COM
WM_KEYDOWN  Handle:1114
```

```
[05/10/31@12:02:43.585-0400] P-005783 T-005735 2 4GL PROEVENTS START COM
CtrlFrame.CSSpin.KeyDown CONTROL-FRAME  Handle:1114 myocx.w T
```

```
[05/10/31@12:03:57.710-0400] P-005783 T-005735 3 4GL PROEVENTS  START GUI
WM_DESTROY FILL-IN customer-name Handle:1126
```

```
[05/10/31@12:02:43.570-0400] P-005783 T-005735 2 4GL PROEVENTS  START COM
WM_CHAR d  Handle:1114
```

```
[05/10/31@12:02:43.570-0400] P-005783 T-005735 2 4GL PROEVENTS  START COM
WM_CHAR TAB  Handle:1114
```

## Format 2: indirect action

When there is an indirect action, logged messages contain the following information:

- **Identifier —** `Action`
- **Description** — A textual description of the indirect action taken

Here are some sample Format 2 log entries:

```
[05/10/31@13:00:00.098-0400] P-000001 T-000002 2 4GL PROEVENTS Action
minimize related windows
```

```
[05/10/31@13:00:00.098-0400] P-000001 T-000002 2 4GL PROEVENTS Action
close related windows
```

```
[05/10/31@13:00:00.098-0400] P-000001 T-000002 2 4GL PROEVENTS Action
restore related windows
```

## Format 3: pushed on to the stack/popped off the stack

When an event is either pushed onto or popped off of the stack, logged messages contain the following information:

- **Identifier —** `Push` or `Pop`.
- **Event name** — For example, `LEAVE`. For a keystroke, the entry includes the key name and associated event, if appropriate.
- **Widget information** — Type (`BUTTON`, for example), name, and handle ID of the object, when available.
- **Procedure name** — Name of the procedure that created the object (not available in all cases).

Here are some sample Format 3 log entries:

```
[05/10/31@13:00:00.098-0400] P-000001 T-000002 2 4GL PROEVENTS PUSH ENTRY
BUTTON button-1 Handle:1564 myproc.p
```

```
[05/10/31@13:00:00.098-0400] P-000001 T-000002 2 4GL PROEVENTS PUSH ENTRY
BUTTON Handle:1987
```

```
[05/10/31@12:19:54.023-0400] P-005805 T-005881 2 4GL PROEVENTS  PUSH CTRL-F
FILL-IN X Handle:1000 test.p
```

```
[05/10/31@12:20:45.054-0400] P-005805 T-005881 2 4GL PROEVENTS  PUSH F6 (NEXT-FRAME)
FILL-IN X test.p
```

```
[05/10/31@13:00:00.098-0400] P-000001 T-000002 2 4GL PROEVENTS POP ENTRY
BUTTON Handle:1987
```

## Format 4: generated by the ABL APPLY statement

When an `APPLY` statement executes, logged messages contain the following information:

- **Identifier —** `APPLY`.
- **Event name** — For example, `LEAVE`. For a keystroke, the entry includes the key name and associated event, if appropriate.
- **Widget information** — Type (`BUTTON`, for example), name, and handle ID of the object, when available.
- **Routine name and line number** — Name and line number of the routine (`.p` or `.w` file or method) and internal procedure, if appropriate, where the `APPLY` statement is. The line number is based on the Debugger listing file.

Here are some sample Format 4 log entries:

```
[05/10/31@13:00:00.098-0400] P-000001 T-000002 2 4GL PROEVENTS APPLY ENTRY
BUTTON button-1 myproc myp.p Line 10
```

```
[05/10/31@13:00:00.098-0400] P-000001 T-000002 2 4GL PROEVENTS APPLY ENTRY
BUTTON Handle:1987 myproc myp.p Line 10
```

### Format 5: trigger execution

When a trigger executes, logged messages contain the following information:

- **Identifier —** `Begin-Trigger` or `End-Trigger`.
- **Additional information** — Can include: the event name, for example, `LEAVE`; object type and name, and handle ID (when available), or internal procedure name for persistent triggers; the word `ANYWHERE`, if it is an `ANYWHERE` trigger. For objects that are no longer valid, the entry also includes the words `object not valid`, as in the case when the window following it has been destroyed.
- **Procedure name** — Name of the routine (`.p` or `.w` file or method) and internal procedure, if appropriate, where the trigger is.

Here are some sample Format 5 log entries:

```
[05/10/31@13:00:00.098-0400] P-000001 T-000002 2 4GL PROEVENTS Begin-Trigger
"LEAVE OF button button-1" Handle:1234 myprog.p
```

```
[05/10/31@13:00:00.098-0400] P-000001 T-000002 2 4GL PROEVENTS End-Trigger
"LEAVE OF button button-1" Handle:1245
```

```
[05/10/31@13:00:00.098-0400] P-000001 T-000002 2 4GL PROEVENTS Begin-Trigger
"LEAVE OF button" RUN myinternalproc Handle:1256 myproc.p
```

```
[05/10/31@13:00:00.098-0400] P-000001 T-000002 2 4GL PROEVENTS Begin-Trigger
"F2 OF FILL-IN" ANYWHERE Handle:1278 myproc.p
```

### Format 6: event discarded

When an event is discarded, logged messages contain the following information:

- **Identifier —** `PRGS`
- **Event name** — For example, `ENTRY`
- **Additional information** — In most cases, the word `discarded`, followed by the reason for not processing the event, if known; for example, `widget disabled`
- **Widget information** — Type (`BUTTON`, for example), name, and handle ID of the object, when available

Here are some sample Format 6 log entries:

```
[05/10/31@13:00:00.098-0400] P-000001 T-000002 2 4GL PROEVENTS PRGS ENTRY
BUTTON b1 Handle:1234 discarded widget disabled
```

```
[05/10/31@13:00:00.098-0400] P-000001 T-000002 2 4GL PROEVENTS PRGS ENTRY
BUTTON Handle:1987 discarded widget disabled
```

```
[05/10/31@12:36:32.226-0400] P-005747 T-005914 2 4GL PROEVENTS  PRGS ESC (END-ERROR)
FILL-IN FILL-IN-1 Handle:1568 discarded
```

```
[05/10/31@12:37:32.226-0400] P-005747 T-005914 2 4GL PROEVENTS  PRGS LEAVE FILL-IN
FILL-IN-1  Handle:1598 discarded return no-apply
```

```
[05/10/31@12:38:32.226-0400] P-005747 T-005914 2 4GL PROEVENTS  PRGS ESC FILL-IN
FILL-IN-1 (END-ERROR) Handle:1599 field validation failed
```

## Sample Event logs

The sample event logs that follow assume a logging level of `2` and a log entry type of `ProEvents.*`.

The next sample event log involves a `RETURN NO-APPLY` statement. It is based on the following scenario:

- There is a window with a fill-in and a button.
- The fill-in has the focus when the user clicks the button.
- The fill-in's `LEAVE` trigger executes a `RETURN NO-APPLY`.
- The AVM discards both the `LEAVE` and `ENTRY` events.

Here is the sample log file (with header suppressed):

```
START GUI WM_LBUTTONDOWN BUTTON BUTTON-1 myprog.p
START GUI WM_KILLFOCUS FILL-IN FILL-IN-1 myprog.p
START GUI WM_SETFOCUS BUTTON BUTTON-1 myprog.p
START GUI WM_LBUTTONUP BUTTON BUTTON-1 myprog.p
PUSH FOCUS-IN BUTTON BUTTON-1 myprog.p
PUSH ENTRY BUTTON BUTTON-1 myprog.p
PUSH LEAVE FILL-IN FILL-IN-1 myprog.p
Begin-Trigger "LEAVE OF FILL-IN FILL-IN-1" myprog.p
End-Trigger "LEAVE OF FILL-IN FILL-IN-1"
PRGS LEAVE FILL-IN FILL-IN-1 return no-apply
POP LEAVE FILL-IN FILL-IN-1 myprog.p
START GUI WM_KILLFOCUS BUTTON BUTTON-1 myprog.p
START GUI WM_SETFOCUS FILL-IN FILL-IN-1 myprog.p
PRGS ENTRY BUTTON BUTTON-1 discarded
POP ENTRY BUTTON BUTTON-1 myprog.p
PRGS FOCUS-IN BUTTON BUTTON-1 discarded
POP FOCUS-IN BUTTON BUTTON-1 myprog.p
```

The next sample Event log shows a common event cycle. It is based on the following scenario:

- There is a window with a toggle box, a fill-in, and a button.
- The toggle box has the focus when the user clicks the button.
- The button's `CHOOSE` trigger runs the `APPLY` statement that applies an event to the fill-in.

Here is the sample log file (with header suppressed):

```
START GUI WM_LBUTTONDOWN BUTTON BUTTON-1 win1.p
START GUI WM_KILLFOCUS FILL-IN FILL-IN-1 win1.p
START GUI WM_SETFOCUS BUTTON BUTTON-1 win1.p
START GUI WM_LBUTTONUP BUTTON BUTTON-1 win1.p
PUSH FOCUS-IN BUTTON BUTTON-1 win1.p
PUSH FOCUS-IN BUTTON BUTTON-1 win1.p
PUSH ENTRY BUTTON BUTTON-1 win1.p
PUSH LEAVE FILL-IN FILL-IN-1 win1.p
Begin-Trigger "LEAVE OF FILL-IN FILL-IN-1" win1.p
End-Trigger "LEAVE OF FILL-IN FILL-IN-1"
POP LEAVE FILL-IN FILL-IN-1 win1.p
Begin-Trigger "ENTRY OF BUTTON BUTTON-1" win1.p
START GUI WM_ENABLE WINDOW  Handle:1129 win1.p
START GUI WM_ENABLE WINDOW  Handle:1029
START GUI WM_KILLFOCUS BUTTON BUTTON-1 win1.p
START GUI WM_SETFOCUS WINDOW  Handle:1129 win1.p
START GUI WM_ENABLE WINDOW  Handle:1129 win1.p
START GUI WM_ENABLE WINDOW  Handle:1029
START GUI WM_KILLFOCUS WINDOW  Handle:1129 win1.p
START GUI WM_SETFOCUS BUTTON BUTTON-1 win1.p
End-Trigger "ENTRY OF BUTTON BUTTON-1"
POP ENTRY BUTTON BUTTON-1 win1.p
POP FOCUS-IN BUTTON BUTTON-1 win1.p
POP FOCUS-IN BUTTON BUTTON-1 win1.p
PUSH CHOOSE BUTTON BUTTON-1 win1.p
Begin-Trigger "CHOOSE OF BUTTON BUTTON-1" win1.p
APPLY ENTRY FILL-IN FILL-IN-1 USER-INTERFACE-TRIGGER win1.p Line 194
PUSH APPLY_ENTRY FILL-IN FILL-IN-1 win1.p
PUSH APPLY_ENTRY FILL-IN FILL-IN-1 win1.p
PUSH ENTRY FILL-IN FILL-IN-1 win1.p
PUSH LEAVE BUTTON BUTTON-1 win1.p
POP LEAVE BUTTON BUTTON-1 win1.p
START GUI WM_KILLFOCUS BUTTON BUTTON-1 win1.p
START GUI WM_SETFOCUS FILL-IN FILL-IN-1 win1.p
POP ENTRY FILL-IN FILL-IN-1 win1.p
POP APPLY_ENTRY FILL-IN FILL-IN-1 win1.p
POP APPLY_ENTRY FILL-IN FILL-IN-1 win1.p
End-Trigger "CHOOSE OF BUTTON BUTTON-1"
POP CHOOSE BUTTON BUTTON-1 win1.p
```

As you can see by examining these samples, there are bursts of logging activity during short spans of user interaction. Thus, it is most efficient to turn on logging at run time, using the `LOG-MANAGER` system handle attributes, perform the interaction you want to trace, and then turn off logging. For more information, see *ABL Reference*.

# File access logging

Specifying the file access (`FileID`) log entry type turns on logging of file access operations (opening and closing files) and file access error messages that do not include the filename. Information is logged at logging level 2 (Basic) and higher.

Some file access error messages do not include the filename in the message; they show only the file ID, which can make it difficult to diagnose the problem. When you turn on file access logging and such an error occurs, OpenEdge includes both the file's name and ID in the log entry. File access logging is also useful in identifying file access errors or inefficiencies in your code (for example, you might be opening or closing a file unnecessarily, setting PROPATH or PATH incorrectly, or exceeding some file limit).

Log entries generated for this type contain the tag FILEID. An example of FileID logging messages (with headers suppressed) is shown:

```
FILEID Open d:\temp\program.r ID=30
FILEID Close d:\temp\program.r ID=30
FILEID Open d:\temp\srt01934 ID=31
FILEID Error 219 for file d:\temp\srt01934 ID=31
```

You can also turn on this logging at run time, by using the LOG-MANAGER system handle. For example, you can include the following line in the ABL code:

```
LOG-MANAGER:LOG-ENTRY-TYPES = "FileID"
```

You can turn off this logging in the ABL application by resetting the attribute. For example:

```
LOG-MANAGER:LOG-ENTRY-TYPES = ?
```

# Ignored operation logging

Specifying the ignored operation (IgnoredOps) log entry type turns on logging of the names and code locations of all window attributes or methods that the AVM ignores on embedded ABL windows. This log entry type has only one logging level, 2 (Basic).

When you embed an ABL window in a .NET control container, many of the window attributes and methods have no function in the .NET environment. ABL ignores any access that you make to these attributes and methods, and this log entry type allows you to identify where you might be using these ignored window elements in your code.

Examples of ignored operations include setting the MAXIMUM-WIDTH attribute and executing the MOVE TO TOP method. For more information about operations that the AVM ignores when they occur in the context of an embedded window, see the EmbeddedWindow property reference entry in *ABL Reference*.

For each ignored operation, OpenEdge logs the source file name, the line number, and the name of the attribute or method, as shown in the following example:

```
[08/03/20@11:12:03.737-0400] P-006124 T-005700 2 4GL IGNOREDOPS (Procedure: 'KeepOnTop
 testwin.w' Line:646) Attribute/Method ALWAYS ON TOP ignored for embedded window
```

**Note:** It is not always possible for the AVM to distinguish between an attribute and a method. Therefore, the log entry specifies "Attribute/Method" as shown in the preceding example.

To create log entries for ignored operations, use the Log Entry Types (`-logentrytypes`) startup parameter:

```
-logentrytypes IgnoredOps
```

You can also turn on this logging at run time, by using the `LOG-MANAGER` system handle. For example, you can include the following line in the ABL code:

```
LOG-MANAGER:LOG-ENTRY-TYPES = "IgnoredOps"
```

You can turn off this logging in the ABL application by resetting the attribute. For example:

```
LOG-MANAGER:LOG-ENTRY-TYPES = ?
```

## NameServer logging

Specifying the NameServer (NSPlumbing) log entry type turns on logging for the NameServer component. Information is logged at the following logging levels:

- **Level 2 (Basic)** — Logs messages when a NameServer starts and stops, load balancing status and messages on registering and unregistering brokers. This is the default logging level.
- **Level 3 (Verbose)** — Same as logging Level 2, plus logs requests from clients, and status information on the communication between client and NameServer and between NameServer and brokers.
- **Level 4 (Extended)** — Same as logging Level 3, plus logs debugging messages on broker unregistering.

Log entries generated for this type contain the tag `NSPlumbing`.

The following example shows log entries for the NameServer component with logging level set to VERBOSE(3) (with headers suppressed):

```
2 NS NSPlumbing Starting NameServer NS1 rmi://test:20931/NS1 Port = 5162 TimeOut = 30.
 (8196)
2 NS NSPlumbing Load Balancing Disabled For NameServer NS1. (8200)
2 NS NSPlumbing Registering Broker 11d1def534ea1be0:283b8a:e8c5ff9235:-7da4 AS.icfrepos
 test/127.0.0.1 3095. (8205)
2 NS NSPlumbing Broker 11d1def534ea1be0:283b8a:e8c5ff9235:-7da4 Registered For
Application Service AS.icfrepos. (8207)
3 NS NSPlumbing Response sent to 127.0.0.1 2837. (8202)
3 NS NSPlumbing Request received from 127.0.0.1 2843 for AS.icfrepos. (8201)
3 NS NSPlumbing AppService = AS.icfrepos Found = true Number Of Brokers = 1. (8206)
3 NS NSPlumbing Response sent to 127.0.0.1 2843. (8202)
3 NS NSPlumbing Known Broker   11d1def534ea1be0:283b8a:e8c5ff9235:-7da4   AS.icfrepos
    test/127.0.0.1 3095. (8209)
```

# Query information logging

Query information logging fills a void for ABL developers trying to evaluate what happens when a query executes and how efficient query execution might be. In large deployed applications, it is impractical for developers to add diagnostics to their code to track and analyze the efficiency and performance of queries.

Query information logging takes advantage of the OpenEdge logging infrastructure to provide you with extended logging information that helps you to locate and correct issues with queries that cause performance degradation (such as poorly indexed queries, or client/server queries that result in excess network traffic).

The following sections provide information about using query information logging.

## Query information log entry type and logging levels

Specifying the query information (`QryInfo`) log entry type enables logging of queries (each open query and `FOR EACH` block) executed in an application. Information is logged at the following logging levels:

- **Level 2 (Basic)** — Logs information about individual queries—this is the default logging level
- **Level 3 (Verbose)** — Logs information about all queries

The following information is logged at each level:

- Index listing
- Query resolution details
- Query statistics

If you set the level to 4 (Extended), information is logged at level 3 (Verbose).

## Starting query information logging

To create query information log entries, use the Log Entry Types (`-logentrytypes`) startup parameter or, for the AppServer and WebSpeed, the s`rvrLogEntryTypes` property in the `ubroker.properties` file. Specify the `QryInfo` log entry type alone or in a comma-separated list that includes other log entry types. For example:

```
-logentrytypes QryInfo
```

```
-logentrytypes QryInfo,ProEvents.UI.Command
```

You can also turn on this logging at run time, using the `LOG-MANAGER` system handle. For example, you can include the following line in the ABL code:

```
LOG-MANAGER:LOG-ENTRY-TYPES = "QryInfo:3"
```

You can turn off this logging in the ABL application by resetting the attribute. For example:

```
LOG-MANAGER:LOG-ENTRY-TYPES = ?
```

Whichever way you start query information logging, you must do so before the query starts, or else no logging will occur for this query. If you turn off query information logging before the query completes, no logging will occur for this query.

## Starting query information logging on individual queries

You can turn on query information logging programmatically for an individual query by doing the following:

- Ensure that the logging level of the `QryInfo` log entry type is Basic (level 2).
- Set the `BASIC-LOGGING` attribute of the query object handle to `TRUE` before the query starts, as follows:
  - For a dynamically opened query, this is before the `QUERY-PREPARE()` method
  - For a statically opened query, this is before the `OPEN QUERY` statement

**Note:** Since `FOR EACH` and static `PRESELECT` queries do not have an associated handle, you cannot log them at level 2.

If the query has already started, no query information logging occurs.

Upon successful startup, OpenEdge writes a message to the log file noting that it has turned on query information logging for the query; the message includes a Query ID, handle, and query name.

To turn off query information logging for the query, set the `BASIC-LOGGING` attribute of the query object handle to `FALSE`. If you do this before the query completes, no query statistics are written. OpenEdge writes a message to the log file noting that it has turned off query information logging for the query; the message includes a Query ID, handle, and query name.

## Writing query statistics on-demand

You can write the currently accumulated query information logging statistics on a query to the log before the query completes, by calling the `DUMP-LOGGING-NOW( )` method of the query object handle.

When you call `DUMP-LOGGING-NOW( )` for a pre-pass query, OpenEdge writes the second set of statistics to the log. OpenEdge writes the first set after it builds the result-list. For more information, see Query statistics for pre-pass queries on page 158.

The `DUMP-LOGGING-NOW( )` method returns `TRUE` if OpenEdge successfully writes the statistics to the log file; `FALSE` otherwise. Logging will fail, for example, if query information logging is turned off (for the individual query or in general), or if the query started before query information logging was turned on.

The syntax for the `DUMP-LOGGING-NOW( )` method is:

```
query-handle:DUMP-LOGGING-NOW(reset)
```

The `DUMP-LOGGING-NOW()` method takes an optional logical argument that, if set to `TRUE`, resets the query's statistics to zero after writing to the log. `FALSE` is the default. In either case, statistics continue to accumulate as the query completes.

The following example writes to the log file after each result and resets the accumulated values:

```
DEFINE VARIABLE qh AS HANDLE NO-UNDO.
CREATE QUERY qh.
qh:SET-BUFFERS(BUFFER customer:HANDLE).
qh:QUERY-PREPARE("FOR EACH customer WHERE customer.custnum < 5"). qh:QUERY-OPEN().
REPEAT:
  qh:GET-NEXT().
  IF qh:QUERY-OFF-END THEN LEAVE.
  qh:DUMP-LOGGING-NOW(TRUE).
END.
```

## Generating query information

The following ABL constructs generate query information logging:

- `FOR [ EACH | FIRST | LAST ]` statements:

```
FOR EACH customer
```

- Static `PRESELECT` statement on `REPEAT` or `DO` blocks:

```
REPEAT PRESELECT FOR EACH customer, EACH order OF CUSTOMER
```

- Statically Opened Queries:

```
OPEN QUERY q FOR EACH customer
```

- Dynamically Opened Queries

  It is possible to dynamically open both dynamic and static queries:

  - For a dynamic query object:

```
DEFINE VARIABLE qh AS HANDLE NO-UNDO.
CREATE QUERY qh.
. . .
qh:QUERY-PREPARE("FOR EACH customer").
qh:QUERY-OPEN().
```

- For a static query object:

```
DEFINE VARIABLE qh AS HANDLE NO-UNDO.
DEFINE QUERY q FOR CUSTOMER.
qh = q:HANDLE.
qh:QUERY-PREPARE("FOR EACH customer").
qh:QUERY-OPEN().
```

Query information logging logs information separately for each instance of the same query. For example, if a program containing a query calls itself recursively, information is logged for each instance of the query.

## Distinguishing between query objects

You typically generate ABL query objects as follows:

- You define static queries. For example:

```
DEFINE QUERY q1 FOR customer.
```

- You create dynamic queries. For example:

```
DEFINE VARIABLE my-query-handle AS HANDLE NO-UNDO.
CREATE QUERY my-query-handle.
```

It also is possible for you to dynamically open a static query; that is, you can define the query statically, but then get a handle to the query, and prepare and open the query as you would a dynamic query. Since query information logging reacts to how a query is prepared and opened, not to how it is defined or created, it is necessary to distinguish between static and dynamic queries, and queries that are statically or dynamically opened.

For example, query information logging logs query preparation statistics for dynamically opened queries, as it happens at run time and can affect the performance of an application. For statically opened queries, the preparation happens at compile time, so there is no impact at run time.

There are some places in query information logging that do relate specifically to dynamic and static queries. For example, dynamic queries typically do not have a name, while static queries always have a name. This means that a dynamically opened query might or might not have a name. Thus, query information logging will log the name only when it is available.

Both database and temp-table queries are logged.

## Unsupported queries

OpenEdge provides no query information logging for SQL statements within an ABL procedure, or for free-standing `FIND` statements. For example:

```
REPEAT:
  FIND NEXT customer.
END.
```

Similarly, query information logging does not log query information for WebSpeed support programs, such as any programs in the `web/objects`, `webutil`, `webedit`, and `webtools` directories.

## Query information logged

Query information logging logs two categories of information:

- Query plan on page 156
- Query statistics on page 157

Since the information in these two categories is not necessarily written contiguously to the log file, a unique identifier links the two. This Query ID, which is a hexadecimal string, has no significance beyond matching a query's plan to its statistics.

## Query plan

The query plan contains the following information about how OpenEdge extracts information for the query:

- Procedure name and line number where the query started:
  - For dynamically opened queries, this is the `QUERY-PREPARE()` method
  - For statically opened queries, this is the `OPEN QUERY` statement
  - For static `PRESELECT` and `FOR` statements, this is the top of the `FOR` or `REPEAT/DO PRESELECT` block.
- Query ID, the query's unique identifier.
- For statically and dynamically opened queries, a string representation of the query handle, and the name of the query (if available).
- Whether the query is a `FOR` statement, a static `PRESELECT` statement, a statically opened query, or dynamically opened query.
  - For a join performed on the server, `Server-side join` is written after the type.
- For dynamically opened queries, the `PREPARE-STRING`.
- When the query was prepared:
  - For `FOR` and static `PRESELECT` statements, and statically opened queries, this is at compile time. Query information logging will report "Prepared at Compile Time".
  - For dynamically opened queries, this is at run time. Query information logging will report "Prepared at Runtime".
- Whether the client is performing sorting.
- Whether this is a `SCROLLING` query, and whether it uses `INDEXED-REPOSITION`.
- Table and index information for each table used in the query. The index information that query information logging provides, such as the `INDEX-INFORMATION` attribute of the query object handle, is already available within ABL. To get more complete index information, use `COMPILE XREF`. For more information on indices, see Identifying indices on page 157.
- Whether the query uses `PRESELECT`. Using the `PRESELECT` keyword in a dynamically opened query has the same effect as the static `PRESELECT`, that is, it builds a preselected list of records.

- The time it takes to prepare dynamically opened queries.

OpenEdge logs and collates query plan information at specific times, based on query type, as follows:

- On a `FOR` statement, after the top of the `FOR` statement block executes
- On a statically opened query, after the `OPEN QUERY` statement executes
- On a dynamically opened query, after the `QUERY-PREPARE()` method of the query handle executes
- On the static `PRESELECT` statement, after the top of the `REPEAT | DO PRESELECT` block executes

### Identifying indices

Query information logging identifies single and multiple index brackets used by the query engine for `FOR` and `PRESELECT` statements. It also identifies indices for dynamic and static queries that use multiple index brackets. Selected indices will display in a comma-separated list in the Query Plan.

---

**Note:** When using dataservers, the logged index behavior may not accurately reflect the indices used to resolve the query against the non-OpenEdge database. For more information, see *Introduction to the Database*.

---

For the following FOR EACH statement:

```
FOR EACH invoice
  WHERE ((invoice.ordernum >= 102) AND
           (invoice.custnum >= 28 AND invoice.custnum <= 30) AND
           (invoice.invoicenum > 100 AND invoice.invoicenum <= 105)) OR
         ((invoice.ordernum > 118 AND invoice.ordernum <= 119) AND
           (invoice.custnum = 46) AND (invoice.invoicenum > 115 AND
           (invoice.invoicenum < 120)) OR
           (invoicenum = 136)
```

Query information logging produces the following Query Plan:

```
Query Plan: C:\work\100a\qitest.p line 1
QueryId: 0xec091c
Type: FOR Statement
Client Sort: N
Scrolling: N
Table: sports2000.Invoice
 Indexes: InvoiceNum,CustNum
```

## Query statistics

Query statistics provide information on the query as it extracts data from the database. That is, on the retrieval of records during the query. This information is intended to help you evaluate query performance.

For the purposes of gathering statistics, queries fall into either of two categories:

- **Pre-pass queries** — Queries that build a result-list completely, in one ABL statement, before going on to the next ABL statement. The AVM builds the result-list at different times for different types of queries. These queries include:

  - Static and dynamic queries that perform sorting or pre-fetching. (The AVM builds a complete result-list only on the first `GET NEXT` statement or `GET-NEXT()` method executed.)
  - Static `PRESELECT` statements. The AVM builds a complete result-list on the `PRESELECT` statement.
  - `FOR` statements that sort on the client and build a list of results in a sort file. The AVM builds a complete result-list on the `FOR EACH` statement.

- **Non-pre-pass queries** — Queries that do not build a result-list completely before executing the next ABL statement. These queries include `FOR EACH` statements and queries that sort by an index, enabling the server to identify the next record from the index, without any sorting by the client.

  Between record retrievals, the client steps through ABL statements in an iterating block, for example, between a `FOR EACH` and its corresponding `END` statement. In cases where the client is selecting the records, only fetches involving records that satisfy selection criteria iterate through the loop.

Query information logging logs different information depending on whether the query is a pre-pass query or a non-pre-pass query. Both pre-pass and non-pre-pass queries write statistics when the query is complete. A query is considered complete at the following points in the code:

- For `FOR` and static `PRESELECT` statements, at the end of the block (the `END` statement). Any statements that cause a premature exit from the block still result in end-of-block processing, and the logging of query statistics.
- For static and dynamic queries, when the AVM closes the query. The following are examples of when the AVM closes the query:
  - On a `CLOSE QUERY` statement or `QUERY-CLOSE()` method
  - On a `OPEN QUERY` statement or `QUERY-OPEN()` method if the query is already open
  - When the query goes out of scope, that is, the program containing the query exits
  - When a program deletes a buffer used by a dynamically opened query

## Query statistics for pre-pass queries

OpenEdge provides two sets of query statistics for pre-pass queries: first, when the AVM builds the result-list; second, when the query is complete (that is, at the end of the block or when the AVM closes the query).

OpenEdge logs the following query statistics when it completely builds the result-list for a pre-pass query:

- Procedure name and line number where the AVM built the result-list.
- Query ID, the query's unique identifier.
- For static and dynamic queries, a string representation of the query handle, and the name of the query object (if available).
- Number of entries in the result-list.
- Time taken to build the result-list. See Time taken to build a result-list on page 161.
- Database block access required to build the result-list. See Database block access on page 161.
- Table and index read access statistics. See Read Access Statistics on page 161.
- Number of records for each table in the query that were sent to the client to resolve the query. See Number of records sent by the server on page 160.
- Field list used to optimize the query. See Field list information on page 161.

- Whether the client rather than the server performs selection of records for each table in the query.
- Number of times the AVM has opened this query. A program can reuse static and dynamic query objects. Opening a query multiple times, as within a loop, might indicate poor programming practice.
- For dynamically opened queries, query preparation statistics. See Query preparation statistics on page 160.

OpenEdge logs the following query statistics when the pre-pass query is complete:

- Procedure name and line number where the block ends or the AVM closes the query.
- Query ID, the query's unique identifier.
- For statically and dynamically opened queries, a string representation of the query handle, and the name of the query object (if available).
- Database block access required to retrieve the ABL records. This is not logged for queries on temp-tables. See Database block access on page 161.
- Table and index read access statistics. See Read Access Statistics on page 161.
- Number of available records for each table in the query. See Number of available records in the query on page 160.
- Whether the program used `INDEX-REPOSITION` on the query.

### Query statistics for non-pre-pass queries

OpenEdge provides a single set of statistics for non-pre-pass queries at query completion. Query information logging logs the following query statistics when the query is complete:

- Procedure name and line number where the block ends or where the AVM closes the query.
- Query ID, the query's unique identifier.
- For dynamic or static queries, a string representation of the query handle, and the name of the query object (if available).
- Database block access required to resolve the query. This is not logged for queries on temp-tables. See Database block access on page 161.
- Table and index read access statistics. See Read Access Statistics on page 161.
- Field list used to optimize the query. See Field list information on page 161.
- Number of available records for each table in the query. See Number of available records in the query on page 160.
- Number of records for each table in the query that were sent to the client to resolve the query. See Number of records sent by the server on page 160.
- Whether the client rather than the server performs selection of records for each table in the query.
- For dynamic or static queries, number of times the AVM has opened this query.
- For dynamically opened queries, query preparation statistics. See Query preparation statistics on page 160.
- For scrolling queries, whether the program used `INDEX-REPOSITION` on the query.

## Query preparation statistics

Query preparation determines how the AVM will access the requested data, for example, which indices to use. The compiler prepares statically opened queries at compile time. The AVM prepares dynamically opened queries at run time, using the `QUERY-PREPARE()` method.

Because preparation of a dynamically opened query can have an impact on performance, query information logging tracks the following statistics on dynamically opened queries:

- Number of times the AVM has prepared this query
- Time taken to prepare the query (in milliseconds)
- Database block access required to prepare the query

## Number of records sent by the server

The server, in resolving a query, sends a number of records to the client. In certain cases, the client determines whether a record satisfies selection criteria. If a query returns a large number of records, only a few of which the client determines satisfy the selection criteria, the query might be inefficient.

For each table in a query, query information logging tracks:

- The number of records sent to the client. For self-service and single-user database connections, this refers to the number of records selected by the database engine.
- The number of records that satisfied the criteria.
- The number of records that failed the criteria.

Optimizations within the client and database server might skew the figures somewhat, as in the cases when the server returns to the client field lists rather than a full record, or a prefetched list of records.

Use these figures with other statistical data to determine if a query is inefficient.

## Number of available records in the query

It is useful to know the number of available records in the query:

- In the context of a `FOR` block or a `REPEAT` loop on a query, this is the number of records the ABL program accessed through a buffer.
- For a browse object on a static or dynamic query, this is the number of records fetched to populate the browse.

Consider this figure in light of other statistical data, such as blocks accessed, or records sent by the server, to determine the efficiency of the query. For example:

- If used in a `FOR` or static `PRESELECT` block with no repositioning, the figure accurately reflects the number of records that satisfy the following search criteria:

```
FOR EACH customer WHERE customer.custnum < 5
```

There are four customers.

- If a program repositions the query, or leaves a block before it fetches all records that satisfy the criteria, the figure is not a true measure of the number of matching records.

### Time taken to build a result-list

It is useful to know how long it takes to build a result-list for a pre-pass query, in evaluating query efficiency. Using this value in combination with other statistical data, such as the number of entries in a result-list, the number of records read, and database block access, you can calculate your own query averages, for example, average time a record is available, or average time for a record to be sent to the client.

### Field list information

Defining field lists in a query improves performance to remote databases and dataservers. The ABL compiler may further optimize the query by adding implicit fields to the field lists or use an implicit field list if the user did not specify one. Query information logging logs all field lists a query uses as part of the Query Statistics. Each field list is logged with the table to which it applies.

For more information on field lists, see *OpenEdge External Programming Interface Reference*.

### Database block access

When evaluating query efficiency, it is useful to consider the number of times the server performs database block accesses on behalf of the client. A disparity in the number of accesses to records returned could indicate poor index selection. OpenEdge tracks database block access on OpenEdge RDBMSs (but not on dataservers) in the `_userio-dbaccess` field of the `_userio` Virtual System Table as follows:

- For pre-pass queries, the number of blocks accessed is the difference between the number of block accesses immediately before the query starts building the result-list, and the number immediately after the query finishes building the list.
- For non-pre-pass queries, and for the reading of ABL records in pre-pass queries, the number of blocks accessed is the accumulation of accesses before and after each record fetch executed by the query.

A large number of database block accesses do not necessarily indicate a poorly conceived query. The number stored in the `_userio-dbaccess` field is a combined figure that includes both RM (data) and index block accesses. Although a high count on index accesses might predictably indicate inefficiencies in index use, a high count on RM accesses might simply mean that records are scattered throughout the database rather than in contiguous blocks. Thus, you have to weigh this value with other statistical data to determine query efficiency.

---

**Note:** Although not recommended, it is possible for a query to access tables in more than one database. For any such queries, OpenEdge records block access statistics for each database accessed. OpenEdge does not, however, track block access statistics for databases in single-user mode.

---

## Read Access Statistics

Query information logging captures table and index read access statistics for each table and index identified in a query against an OpenEdge database. Read access statistics includes reads by all users connected to the database.

---

**Note:** Query information logging omits read access statistics for DataServers and temp-tables. Additionally, read access statistics are not logged for preparing a dynamic query.

---

Table read access statistics provide the number of times read access has occurred on a table. Use Base Table (`-basetable`) with Table Range Size (`-tablerangesize`) to specify the range of tables for which you want to collect statistics. Table statistics are stored in the `_TableStat-Read` field of the `_TableStat` Virtual System Tables (VST).

Index read access statistics provide the number of times read access has occurred on an index. Use Base Index (-baseindex) with Index Range Size (-indexrangesize) to specify the range of indexes for which you want to collect statistics. Index statistics are stored in the _IndexStat-Read field of the _IndexStat VST.

Read access statistics information is located in the Query Statistics after the DB Block access statistics. For each table in the query, or a non-pre-pass query, the client writes the read access statistics in the Query Statistics once the query completes. For a pre-pass query, the client writes the read access statistics in the Query Statistics for building the results list, and again once the query completes. This differentiates the read accesses required to build the result list from those required to iterate through the list.

In order to get the table and index read statistics, the user must ensure the queried tables and indices are within the ranges specified by the startup parameters Base Table (-basetable)/Table Range Size (-tablerangesize) and Base Index (-baseindex)/Index Range Size (-indexrangesize) ranges. If your query accesses tables or indices outside these ranges, query information logging cannot determine the access statistics for the tables or indices, and displays "UNAVAILABLE" for the read access statistics.

For more information on the referenced startup parameters, see *Startup Command and Parameter Reference*. For more information on VSTs, see *Manage the OpenEdge Database*.

## Sample query information logs

Sample output formats (with headers suppressed) of query information logs are shown.

The following is a code example for a non-pre-pass query:

```
FOR EACH invoice
  WHERE ((invoice.ordernum >= 102) AND
         (invoice.custnum >= 28 AND invoice.custnum <= 30) AND
         (invoice.invoicenum > 100 AND invoice.invoicenum <= 105)) OR
        ((invoice.ordernum > 118 AND invoice.ordernum <= 119) AND
         (invoice.custnum = 46) AND
         (invoice.invoicenum > 115 AND invoice.invoicenum < 120)) OR
        (invoicenum = 136),
  EACH order WHERE order.ordernum = invoice.ordernum:
    DISPLAY invoice.invoicenum order.ordernum.
END.
```

The following is an example of the query plan for a non-pre-pass query:

```
Query Plan:  qisample1.p line 1
QueryId: 0x159e004
Type: FOR Statement
Client Sort: N
Scrolling: N
Table: sports2000.Invoice
  Indexes: InvoiceNum,CustNum
Table: sports2000.Order
  Indexes: OrderNum
```

The following is an example of the query statistics for a non-pre-pass query:

```
Query Statistics:  qisample1.p line 11
QueryId: 0x159e004
DB Blocks accessed:
 sports2000 : 43
DB Reads:
 Table: sports2000.Invoice : 11
 Index: Invoice.InvoiceNum : 8
 Index: Invoice.CustNum : 6
 Table: sports2000.Order : 3
 Index: Order.OrderNum : 6
sports2000.Invoice Table:
 4GL Records: 3
 Records from server: 3
  Useful: 3
  Failed: 0
 Select By Client: N
sports2000.Order Table:
 4GL Records: 3
 Records from server: 3
  Useful: 3
  Failed: 0
 Select By Client: N
```

The following is a code example for a pre-pass query:

```
FOR EACH invoice
  WHERE ((invoice.ordernum >= 102) AND
         (invoice.custnum >= 28 AND invoice.custnum <= 30) AND
         (invoice.invoicenum > 100 AND invoice.invoicenum <= 105)) OR
        ((invoice.ordernum > 118 AND invoice.ordernum <= 119) AND
         (invoice.custnum = 46) AND
         (invoice.invoicenum > 115 AND invoice.invoicenum < 120)) OR
        (invoicenum = 136),
  EACH order WHERE order.ordernum = invoice.ordernum BY invoice.custnum:
    DISPLAY invoice.invoicenum order.ordernum.
END.
```

The following is an example of the query plan for a pre-pass query:

```
Query Plan:  qisample2.p line 1
QueryId: 0x159e006
Type: FOR Statement
Client Sort: Y
Scrolling: N
Table: sports2000.Invoice
  Indexes: CustNum,InvoiceNum
Table: sports2000.Order
  Indexes: OrderNum
```

The following is an example of the query statistics for building the result-list on a pre-pass query:

```
QueryId: 0x159e006
Entries in result list: 3
Time to build result list (ms): 16
DB Blocks accessed to build result list:
 sports2000 : 38
DB Reads to build result list:
 Table: sports2000.Invoice : 15
 Index: Invoice.CustNum : 16
 Index: Invoice.InvoiceNum : 2
 Table: sports2000.Order : 0
 Index: Order.OrderNum : 0
sports2000.Invoice Table:
 Records from server: 3
  Useful: 3
  Failed: 0
 Select By Client: N
sports2000.Order Table:
 Records from server: 0
  Useful: 0
  Failed: 0
 Select By Client: N
```

The following is an example of the second query statistics (record retrieval) on a pre-pass query:

```
Query Statistics:  qisample2.p line 12
QueryId: 0x159e006
DB Blocks accessed:
 sports2000 : 15
DB Reads:
 Table: sports2000.Invoice : 3
 Index: Invoice.CustNum : 0
 Index: Invoice.InvoiceNum : 0
 Table: sports2000.Order : 3
 Index: Order.OrderNum : 6
sports2000.Invoice Table:
 4GL Records: 3
sports2000.Order Table:
 4GL Records: 3
```

The following is an example of the query statistics when the read access statistics for the table and index are outside of range:

```
Query Statistics:  idxnum.p line 6
QueryId: 0x159f3ec
DB Blocks accessed:
  sports2004 : 272
DB Reads:
  Table: sports2004.item : UNAVAILABLE  Index: item.Item_Obj : UNAVAILABLEsports2004.item
 Table:
  4GL Records: 134
  Records from server: 134
   Useful: 134
   Failed: 0
Select By Client: N
```

# SAX parser logging

Specifying the SAX parser (SAX) log entry type turns on logging for the SAX parser. The following table lists the message logged as the SAX parser executes.

**Table 43: SAX parser logging messages**

| When logging occurs | What is logged |
|---|---|
| The AVM initializes the parser. | `Initialize parser` |
| The AVM finishes using the parser. | `Exiting parser` |
| The AVM finds a callback to run. | `Callback` *`internal-routine-name`* `found in super procedure`<br>— or —<br>`Callback` *`internal-routine-name`* `found` |
| The AVM does not find a callback to run. | `Callback` *`internal-routine-name`* `not found` |
| For each parameter in a callback, the log message includes the following, depending on the parameter type:<br>• `CHARACTER, MEMPTR` (which contains a string), or `INTEGER`<br>• If `CHARACTER` parameter is empty<br>• `HANDLE` | `parameter` *`parameter-number`*`: value`<br>`parameter` *`parameter-number`*`: <null>`<br>`parameter` *`parameter-number`*`: <valid-handle>` |

The following example shows log entries for the SAX parser at logging level 2 (Basic) with headers suppressed:

```
1 4GL -- Log entry types activated:SAX
2 4GL SAX    Initialize parser
2 4GL SAX    Callback StartDocument not found
2 4GL SAX    Callback StartElement found
2 4GL SAX        parameter 1: <null>
2 4GL SAX        parameter 2: custorders
2 4GL SAX        parameter 3: custorders
2 4GL SAX        parameter 4: <valid handle>
2 4GL SAX    Callback Characters found
2 4GL SAX        parameter 1:
2 4GL SAX        parameter 2: 1
2 4GL SAX    Callback EndElement found
2 4GL SAX        parameter 1: <null>
2 4GL SAX        parameter 2: city
2 4GL SAX        parameter 3: city
2 4GL SAX    Callback EndDocument not found
2 4GL SAX    Exiting parser
```

# Temp-table logging

The temp-table log entry type enables specific logging for temp-tables so that application developers and technical support engineers can gather more information about the temp-tables used by an application. The temp-table entry type provides the following information:

- temp-table name
- temp-table unique ID
- Number of indexes created for a temp-table
- Number of times the `EMPTY TEMP-TABLE` method executed
- Procedure or class name and line where temp-table was created, deleted or emptied
- Number of records, and record size.

A temp-table is defined at compile-time using the `CREATE TEMP-TABLE` statement. The temp-table logging only logs an entry once the dynamic temp-table is prepared, that is, when all its schema definitions are known and when the actual temp-table is created. The `CREATE TEMP-TABLE` statement does not create the actual temp-table at once, but only when the temp-table is prepared.

The Temp-tables log entry type logs information at the following logging levels:

- Level 2 (Basic)
- Level 3 (Verbose)
- Level 4 (Extended)

The log entry type is available to any ABL client, including the AppServer servers, WebSpeed agents and interactive and batch ABL clients.

The format of the log entries at the Basic logging level are:

Creating a temp-table:

```
Created TEMP-TABLE <name> (ID: <id> [DYNAMIC] [NO-UNDO] [GLOBAL] [SHARED] Indexes
<num-indexes>) <proc-info>
```

Deleting a temp-table:

```
Deleted TEMP-TABLE <name> (ID: <id> [DYNAMIC]) <proc-info>
```

Emptying a temp-table

```
Emptied TEMP-TABLE <name> (ID: <id>) <proc-info>
```

### Example

The following is an example of log entries for the temp-table logging:

```
Created TEMP-TABLE mytt (ID:29 Indexes:3) myTables.p
Created TEMP-TABLE ttCustNoUndo (ID:34 NO-UNDO Indexes:5) myTables.p
Created TEMP-TABLE BeforeOrderNoUndo (ID:37 NO-UNDO Indexes:1) myTables.p
Created TEMP-TABLE ttCustDyn (ID:40 DYNAMIC NO-UNDO Indexes:1) myTables.p @ 41
Emptied TEMP-TABLE ttCustDyn (ID:40) myTables.p @ 50
Deleted TEMP-TABLE ttCustDyn (ID:40 DYNAMIC) myTables.p @ 59
Deleted TEMP-TABLE BeforeOrderNoUndo (ID:37) myTables.p @ 60
Deleted TEMP-TABLE ttCustNoUndo (ID:34) myTables.p @ 60
Deleted TEMP-TABLE mytt (ID:29) myTables.p @ 60
```

The format of the log entries is:

`Cached TEMP-TABLE <name> (ID: <id>) <object-info>`

`Reused TEMP-TABLE <name> (ID: <id>) <object-info>`

where `<object-info>` is the name of the object oriented object.

## Unified Broker logging

Specifying a Unified Broker log entry type turns on logging for the Unified Broker component. These are the Unified Broker log entry types and the information they provide based on logging level:

- **`UBroker.Basic`** — Logs messages about broker and server startup and shutdown, NameServer registration, and connections from clients at Level 2 (Basic). At Level 3 (Verbose), logs information about all the property values read from the `ubroker.properties` file for that broker instance, and more information on the connection from clients.

  The following example shows log messages for the `Ubroker.Basic` entry type (with headers suppressed):

```
2 UB Basic      ubroker version: v100B (28-Apr-04) (8038)
2 UB Basic      Started listener thread: L-3090 (8043)
3 UB Basic      installDir          : d:\dlc100b
3 UB Basic      properties file     : d:\dlc100b\properties\ubroker.properties
3 UB Basic      Broker              : asbroker1
3 UB Basic      serverType          : AS
```

- **`UBroker.ThreadPool`** — Logs messages about the pool of threads managed by the broker. These threads are used to control the client requests and the server processes. This entry type will log messages when adding, removing, and communicating with these threads. Information is logged at logging level 2 (Basic) and higher.
- **`UBroker.AutoTrim`** — Logs messages about automatic server thread trimming by the broker based on the `autoTrimTimeout` property. Information is logged at logging level 2 (Basic) and higher.
- You should turn on the following log entry types only if Progress Software Corporation Technical Support requests that you do so:
  - **`UBroker.ClientFSM`** — Logs messages about the state of the client thread as it processes the requests, from the moment it receives a connection request to the end of the request. Information is logged at logging level 2 (Basic) and higher.

- **UBroker.ServerFSM** — Logs messages about the state of the server thread as it processes the requests, from the moment it receives a connection request to the end of the request. Information is logged at logging level 2 (Basic) and higher.
- **UBroker.ClientMsgStream** — Logs messages about messages exchanged between the broker and client threads in binary format. Information is logged at logging level 2 (Basic) and higher.
- **UBroker.ServerMsgStream** — Logs messages about messages exchanged between the broker and server threads in binary format. Information is logged at logging level 2 (Basic) and higher.
- **UBroker.ClientMsgQueue** — Logs messages about the client thread as it processes queued messages. Information is logged at logging level 2 (Basic) and higher.
- **UBroker.ServerMsgQueue** — Logs messages about the server thread as it processes quened message. Information is logged at logging level 2 (Basic) and higher.
- **UBroker.ClientMemTrace**— Traces messages being processed by the client threads. Information is logged at logging level 2 (Basic) and higher.
- **UBroker.ServerMemTrace** — Traces messages being processed by the server threads. Information is logged at logging level 2 (Basic) and higher.
- **UBroker.Stats** — Logs statistics about the number of requests, plus maximum, minimum, and average request wait and duration times. Information is logged at logging level 2 (Basic) and higher.
- **UBroker.All** — Logs all `UBroker` entry types.

# Web Services Adapter (WSA) and Web Service logging

The WSA log file and Web service SOAP fault response messages provide Web service error information. You control the amount of information provided using the `loggingLevel`, `serviceLoggingLevel`, and `serviceFaultLevel` properties.

## WSA Log entry types

The WSA instance logs error messages in the *instance-name*`.wsa.log` file. This file resides in your OpenEdge working directory. Following are the WSA log entry types and the information they provide based on logging level:

- **4GLProvider** — `4GLProvider` logs messages about locating the deployed Progress service and then invoking it. This includes any verification that the service can execute the deployed AppServer procedure. Level 2 (Basic) logs information if there is any error is in the SOAP request execution. Level 3 (Verbose) logs errors in the SOAP request. Level 4 (Extended) logs information that can be used for debugging, like creating a new `ProcAppObject` or `SubAppObject`, method return parameters, and release of Object ID.
- **Actional** — `Actional` logs messages about creating client interaction and SOAP payloads for WSA. Level 2 (Basic) logs information about the required parameters for the WSDL document listings, WSDL authorization, and the invalid provider type. Level 2 (Extended) logs information about fault messages and raw SOAP responses.Level 4 (Extended) logs information that can be used for debugging, like creation of new `ProcAppObject` or `SubAppObject`, method return parameters, release of Object ID, and such related information.
- **AdminProvider** — `AdminProvider` logs information related to runtime properties and statistical information while managing a deployed WSA. Level 2 (Basic) logs the name of the deployed WSA, the ID of the un-deployed WSA, information about the WSA deployment, and the resetting of the runtime statistics for the WSA. Level 3 (Verbose) logs information related to the application initialization such as runtime properties,

application descriptors, deployment descriptors, and pool managers. It also registers information related to error during SOAP request initialization and execution, and runtime statistics of the WSA. Level 4 (Extended) logs information about error during SOAP request initialization and execution.

- **BrokerClient —** `BrokerClient` maintains logs information related to AppServer connections and remote request communications context information for the construction of a component logging environment. Level 2 (Basic) logs information about the ubroker client.
- **DynamicApi —** `DynamicApi` logs information related to the marshalling and unmarshalling of remote procedure parameters supplied by the 4GLProvider and passed by the BrokerCleint objects to the AppServer to send and receive data from the ABL server. Level 4 (Extended) logs basic error information about a broker reference of a client for the ABL server connection.
- **MsgDebug —** `MsgDebug` logs information related to the HTTP request to the WSA servlet. Level 3 (Verbose) logs information related to the servlet request parameters. Level 4 (Extended) logs information related to the address of the client request to the WSA servlet, and the SOAP request and response. For tracking a specific WSA client IP addresses user must set the debugClients property appropriately in the ubroker.properties file.
- **PoolMgmt —** `PoolMgmt` logs information related to the management of the BrokerClient session pool. Level 4 (Extended) logs information related to the session pool reference for `WSAAppObjects`, `WSASubAppObjects`, and `WSAProcObjects`. This also logs information related to the cleaning up of the bad sessions from the session pool.
- **Properties —** MsgDebug logs information related to the initialization parameters in the `web.xml` file, properties in `ubroker.properties` file, Actional settings, and other optional properties. Level 3 (Verbose) logs the following information:

**Table 44: Information logged for each type of action**

| File | Properties |
|---|---|
| web.xml | `installDir`<br>`propertyFileName`<br>`instanceName` |

| File | Properties |
|---|---|
| ubroker.properties | logFile<br>loggingLevel<br>logAppend<br>logEntryTypes<br>logThreshold<br>numLogFiles<br>webAppEnabled<br>adminEnabled<br>httpErrorPage<br>enableWsdlListings<br>enableWsdl<br>adminAuth<br>wsdlAuth<br>appAuth<br>adminRoles<br>wsdlListingPage<br>noWsdlPage<br>faultLevel<br>wsaUrl<br>appProtocol |
| Actional settings | actionalEnabled<br>actionalGroup<br>actionalPayloadCapture |
| Optional properties | debugClients<br>appProtocol<br>logMsgThreshold |

- **PoolMgmt** — RefCounts logs information related to the connection reference count. The level 4 (Extended) logs the connection references count for a particular session ID.
- **RunProcs** — RunProcs logs information related to the running of an external procedure. The level 4 (Extended) logs the name of an external procedure invoked by a client request.
- RunProcs — RunProcs logs information related to the running of an external procedure. The level 4 (Extended) logs the name of an external procedure invoked by a client request.
- **SessionPool** — SessionPool logs information related to the management of the session pool. The level 4 (Extended) logs information related to the cleaning up of the bad sessions from the session pool.

- **SOAPProc** — SOAPProc provides information about the SOAP request and response.Level 2 (Basic) logs information about the corrupt web application and the errors while locating a service provider. Level 3 (Verbose) logs information related to errors related to unmarshalling of SOAP method parameters, SOAP envelope, and SOAP body, locating the WSA, the SOAP request initialization and execution, availability of a service, and the failed authentication to the service. Level 4 (Extended) logs the complete stacks trace if the Progress level logging is turned on. This might cause duplication of the log file.
- **WSA** — WSA provides information about the processing of SOAP RPC request from an HTTP client.Level 2 (Basic) logs information related to the HTTP request status and the user's access to a particular service.Level 3 (Verbose) logs information related to the WSA request ID, the WSA initialization parameters, the absolute directory path for the WSA instance, the status of the WSA registry, the XML parsing errors, user's access to a particular service, and the errors in SOAP parameters.Level 4 (Extended) logs information related to validation of users and their roles to access the WSA, and the WSA administrator access to the undefined methods.
- **WSAObject** — WSAObject logs information related to the management of the session pool reference. Level 4 (Extended) logs information about the creation of new sessions, adding a new session reference to the session pool, deleting a session reference from the session pool, disconnecting the connection to the server, external non-persistent procedure calls through the session pool reference, the count of the connection references in the session pool, and the cleaning up of a bad session from the session pool.
- **WSAObjPool** — WSAObjPool logs information related to the proxy session pool. The level 4 (Extended) logs information about the proxy session pool, the proxy objects, the AppObject, SubAppObject, and ProcObject.
- **WSDLDoc** — WSDLDoc provides information regarding the request for WSDL document through SOAP engine. The level 2 (Basic) logs information on whether the method call has the correct parameters passed. The level 3 (Verbose) logs information related to conditions to get the WSDL document access through the SOAP engine. The conditions include whether SOAP engine is running, the access to the WSDL document is enabled, and whether the users have the required authorization and authentication. The level 4 (Extended) logs information about error during WSDL document retrieval. It also logs the complete stack trace if the Progress level logging is turned on. This might cause duplication of the log file.

The WSADefault log entry type provides server-related messages for a single WSA instance based on the following logging levels:

- WSADefault — WSADefault Logs processing requests and very severe errors. Use Level 2 in a production environment. Level 3 (Verbose) logs general access problems, startup property values, user authorization errors, request IDs, HTTP error responses, and WSDL retrieval errors. Use Level 3 in a development environment. Level 4 (Extended) Logs HTTP and SOAP requests and responses. Use Level 4 to debug.

**Note:** During production, scan the background event counters looking for errors. Raise the logging level to 3 or 4 to evaluate the errors and determine the problem. When the problem is resolved, return the logging level to 2.

## Setting the WSADefault log entry type

Specifying the logging for Web Services Adapter (WSA) provides information based on logging level. The WSADefault log entry type provides server-related messages for a single WSA instance. You can set the WSADefault log entry type by using OpenEdge Management or OpenEdge Explorer, or directly editing the ubroker.properties file.

## Setting the loggingLevel property

You can set the `loggingLevel` property using OpenEdge Management or OpenEdge Explorer, the `wsaman` utility, or directly editing the `ubroker.properties` file. This persistent value takes affect when the WSA is started. You can also set a temporary value for the `loggingLevel` property.

**Note:** It is not necessary to stop and restart the WSA to change the logging level.

## Setting the serviceLoggingLevel property

The Web service also logs error messages in the *instance-name*`.wsa.log` file. The `serviceLoggingLevel` property affects logged information for a single deployed Web service. You can set the `serviceLoggingLevel` property in the Web service properties file (`*.props`) or by using the `wsaman` utility. This setting takes effect immediately at run time. The `serviceLoggingLevel` property provides information at the following logging levels:

- **Level 2 (Basic)** — Logs severe errors that affect the Web service as a whole and user authentication errors. Use Level 2 when the Web service is running in production.

**Note:** During production, raise the logging level of only the Web Service being debugged.

- **Level 3 (Verbose)** — Logs Web service request execution, SOAP message processing errors, and user authentication success. Use this value during Web service development to provide more information for each Web service request error.
- **Level 4 (Extended)** — Logs error detail and request parameter values. Use Level 4 in a development environment.

## Setting the serviceFaultLevel property

Errors while processing a Web service client request generate a SOAP fault message. The amount of information returned from SOAP fault messages is set by the `serviceFaultLevel` property for the Web service. You specify the amount of information provided with SOAP fault messages by setting the `serviceFaultLevel` property for a single Web service. You set the `serviceFaultLevel` property in the Web service properties file (`*.props`) or by using the `wsaman` utility. This setting takes effect immediately at run time. The `serviceFaultLevel` property provides information at the following logging levels:

- **Level 2 (Basic)** — Logs general error information. Use Level 2 when the Web service is running in production.
- **Level 3 (Verbose)** — Logs Web service request execution, SOAP message processing errors, and user authentication success. Use Level 3 when the Web service is in development or when more detail is required while running in production.

**Note:** All Web service logging and fault levels are controlled individually for each Web service to further reduce processing overhead.

## Understanding Web service information in WSA logs

When the WSA starts up, each WSA instance receives a unique, alphanumeric identifier called the *WSA ID*. The WSA ID appears in the log file after the "ID", as follows:

```
ID f83c04d3ca9b076a:648016:1067934305:-7fff
```

When the WSA instance receives a request from a Web service, the WSA instance assigns the request a unique alphanumeric identifier called *request ID*. When the WSA instance logs information about a Web service request, the WSA includes the request ID in the following format, where *wsaid-value* is the WSA ID and *reqid-value* is the request ID:

**Table 45:**

```
(reqid:wsaid-value#reqid-value)
```

The client uses the WSA ID and the request ID to find more detailed information in the corresponding entries of the WSA log file.

The following log (with headers suppressed) was generated for WSADefault entry type at logging level 2 (Basic):

```
1 wsa1 Starting WSA adapter wsa1, version 10.1A1) ID
f83c04d3ca9b076a:648016:1067934305:-7fff (10620)
1 wsa1 Runtime properties set for WSA
2 wsa1 Admin-Provider Deploying Web Service application: GetCustName
GetCustName 4GL-Provider    Error in SOAP request exectuion: Connect Fauilure:
Application Service GetCustName Unknown at NameServer at Host localhost Port 5162.
(8234) (10926)
```

For more information on Web Services Adapter logging see *Develop Web Services for OpenEdge*.

# WebSpeed Messenger logging

Specifying the WebSpeed Messengers (MsgrTrace) log entry type turns on logging for the WebSpeed Messengers. Information is logged at the following logging levels:

- **Level 2 (Basic)** — For CGIIP, logs the start and end of the request, and the values of the CONTENT_LENGTH, PATH_INFO, and QUERY_STRING variables. For other Messengers, logs the environment variables.
- **Level 3 (Verbose)** — Same as logging level 2, plus the CGIIP environment variables, and the environment variables in binary format for other Messengers.
- **Level 4 (Extended)** — Same as logging level 3, plus the environment variables in binary format for CGIIP. Logging level 4 shows all messages generated at all logging levels.

The following log (with headers suppressed) was generated for `CGIIP` with the `MsgrTrace` entry type at logging level 4(EXTENDED):

```
3 ENV  ------- CONTENT_LENGTH=0
3 ENV  ------- GATEWAY_INTERFACE=CGI/1.1
3 ENV  ------- HTTP_ACCEPT=image/gif, image/x-xbitmap, image/jpeg, image/pjpeg,
application/vnd.ms-excel, application/vnd.ms-powerpoint, application/msword,
application/x-shockwave-flash, */*
3 ENV  ------- HTTP_ACCEPT_LANGUAGE=en-us
3 ENV  ------- HTTP_CONNECTION=Keep-Alive
3 ENV  ------- HTTP_HOST=test
3 ENV   ------- HTTP_USER_AGENT=Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1;
Q312461; .NET CLR 1.0.3705; .NET CLR 1.1.4322)
3 ENV  ------- HTTP_ACCEPT_ENCODING=gzip, deflate
3 ENV  ------- HTTPS=off
3 ENV  ------- SCRIPT_NAME=/scripts/cgiip.exe
3 ENV  ------- SERVER_PORT=80
3 ENV  ------- SERVER_PORT_SECURE=0
3 ENV  ------- SERVER_PROTOCOL=HTTP/1.1
3 ENV  ------- SERVER_SOFTWARE=Microsoft-IIS/5.1
2 ENV  ------- CONTENT_LENGTH= 0
2 ENV  ------- PATH_INFO= /WService=wsbroker1/workshop
2 CGI  TERM-RQ End Request
```

# 7

# Other Diagnostics Resources

This chapter is a resource for information about other diagnostics tools and aids available in the OpenEdge product suite and supporting product documentation.

For details, see the following topics:

- Abnormal termination dump files
- How the AVM handles failure conditions
- Run-time diagnostics
- Analyzing procedures and queries

## Abnormal termination dump files

An abnormal termination can result in some combination of the following dump files:

- `procore` file
- `protrace` file
- `core` file

**Important:** To analyze the contents of a `protrace` file, you need the assistance of Progress Software Corporation Technical Support.

## procore file

A `procore` file is a text file that contains:

- A list of recent ABL (Advanced Business Language) messages, by message number
- An ABL procedure stack trace that includes line numbers from the debug listing file within each procedure

OpenEdge writes the `procore` file to a temporary directory, if you specify the Temporary Directory (`-T`) startup parameter; otherwise, OpenEdge writes the `procore` file to the current working directory. Refer to Procedures and triggers on page 28.

### Sample procore file

```
_12/18/01 13:44:57  [784]
Progress Recent Message(s): (5635) (5407) (5407) (5407) (5407) (5407) (5407) (5407)
(5407) (5407)
  *** 4GL Call Stack ***
4: foo InfiniteLoop.p (InfiniteLoop.p)
4: foo InfiniteLoop.p (InfiniteLoop.p)
4: foo InfiniteLoop.p (InfiniteLoop.p)
4: foo InfiniteLoop.p (InfiniteLoop.p)
4: foo InfiniteLoop.p (InfiniteLoop.p)
4: foo InfiniteLoop.p (InfiniteLoop.p)
... (many more of these) ...
1: InfiniteLoop.p (InfiniteLoop.p)
1: D:\pcspot\dir1\p89453r.ped (D:\pcspot\dir1\p89453r.ped)
588: adecomm/_runcode.p (adecomm/_runcode.r)
3796: ExecuteRun adeedit/_proedit.p (adeedit/_proedit.r)
10550: RunFile adeedit/_proedit.p (adeedit/_proedit.r)
2323: USER-INTERFACE-TRIGGER adeedit/_proedit.p (adeedit/_proedit.r)
11727: adeedit/_proedit.p (adeedit/_proedit.r)
453: _edit.p (\\ruddles\i_91d_nt\dlc\gui\_edit.r)
```

This resulted from running a program in the Procedure Editor that contains an infinite loop. Line numbers appear at the beginning of a line, followed by a colon.

## protrace file

When an OpenEdge process terminates abnormally, OpenEdge automatically creates a text file named `protrace` in the current directory. You can also force creation of the `protrace` file for a running program by using a command line utility (see Generating a protrace file for a running program on page 188). To analyze the contents of a `protrace` file, you need the assistance of Progress Software Corporation Technical Support.

The `protrace` file contains:

- A stack trace of the ABL Virtual Machine (AVM) or other OpenEdge process (such as a database server or utility) that terminated abnormally
- A stack trace of the executing ABL program itself, where applicable
- A list of the persistent procedures or classes that are in memory, with their handle IDs
- A list of startup parameters in effect for the ABL session
- Symbolic information (when available)

Refer to the sample Procedures and triggers on page 28.

### Sample protrace file

```
PROGRESS stack trace as of Tue Jan  6 13:15:16 2009
Command line arguments are
/dlc/bin/_progres -1 -p test.p
Startup parameters:
-pf /dlc/startup.pf,(end .pf),-1,-p test.p
(1) uttrace() +0x134 from: /dlc/bin/_progres
(2) drsigs() +0x6e4 from: /dlc/bin/_progres
(3) drExitOnTerm() +0x30c from: /dlc/bin/_progres
(4) drExitOnTerm() +0x558 from: /dlc/bin/_progres
(5) _setuid() +0x60 from: /usr/lib/libc.so.1
(6) _select() +0x2ec from: /usr/lib/libc.so.1
(7) rnSocketSelect() +0x12c from: /dlc/bin/_progres
(8) wvResetCurItem() +0x1e4 from: /dlc/bin/_progres
(9) wvReadKey() +0x8c from: /dlc/bin/_progres
(10) ioHandlePause() +0x338 from: /dlc/bin/_progres
(11) iopause() +0x108 from: /dlc/bin/_progres
(12) rniopaus() +0x230 from: /dlc/bin/_progres
(13) rnexec_entry() +0x618 from: /dlc/bin/_progres
(14) rninterpret() +0x54 from: /dlc/bin/_progres
(15) rnrq() +0x120 from: /dlc/bin/_progres
(16) main() +0x410 from: /dlc/bin/_progres
(17) _start() +0x108 from: /dlc/bin/_progres
** ABL Stack Trace **
--> myproc test.p (test.p) at line 4
** Persistent procedures/Classes **
```

## core file

A `core` file is specific to UNIX. It is a binary file that requires a UNIX debugger such as `dbx`. A `core` file contains:

- A stack trace of the AVM executing ABL code
- Hex representation of the parameters to each routine

On some systems, an executable's permissions do not allow `core` file generation.

# How the AVM handles failure conditions

The following table summarizes the types of failures you might encounter and the diagnostic information that results.

**Table 46: Program failures and resulting dump files**

| This type of failure | Provides this diagnostic information |
|---|---|
| The AVM detects a fatal system error, such as an out-of-memory condition. | A `procore` file. Additionally, if the message is marked as coredump:<br>• **Windows** — `protrace` file<br>• **UNIX** — `protrace` and `core` files |

| This type of failure | Provides this diagnostic information |
|---|---|
| An exception occurs in OpenEdge Java code (for example, broker, NameServer, WebSpeed Messenger). | A stack trace written to the log file. The broker continues to run for threads associated with other clients. |
| There is a memory violation. | • **Windows** — `protrace` file; `Dr. Watson` log, if set in the registry<br>• **UNIX** — `procore`, `protrace`, and `core` files |
| OpenEdge executable hangs; you must terminate it using the `kill` command or the Task Manager. | • **Windows** — None (usually, waiting for an event that never occurs in an infinite loop; or a GPF occurs that hangs during failure processing)<br>• **UNIX** — `protrace` file, if you send a `SIGUSR1` to the process as part of the `kill` operation |

# Run-time diagnostics

This section identifies various types of information that you can capture while the application is executing. Developers and technical support personnel should find this information useful.

The following example shows sample code `execlog.p`.

## Sample executable procedure

```
/* execlog.p */
DEFINE VARIABLE ix AS INTEGER NO-UNDO.
FOR EACH customer FIELDS (name) NO-LOCK BY customer.name:
  DISPLAY customer.name.
  ix = ix + 1.
END.
RUN internalA (ix).
RUN execlog2.p (OUTPUT ix).
DISPLAY ix.
QUIT.
PROCEDURE internalA:
  DEFINE INPUT PARAMETER ii AS INTEGER NO-UNDO.
  DISPLAY "Number of Customers is " ii.
END.
/* execlog2.p */
DEFINE OUTPUT PARAMETER ix AS INTEGER NO-UNDO.
DO ix = 10 TO 20:
  DISPLAY ix.
END.
```

## Collecting r-code usage statistics

When you use statistics startup parameters, OpenEdge writes statistics at the end of the session. You can use the SHOW-STATS statement in concert with the startup parameters to write statistics at specific times during execution. The statistics collected with the SHOW-STATS statement do not include the report of all startup parameters and their values that you get with the Statistics (-y) startup parameter.

Statistics-gathering mechanisms include these various startup parameters:

- **Statistics ( -y )** — Use this startup parameter to collect procedure access and usage statistics throughout the OpenEdge session. This parameter also sends a report of all startup parameters and their values to the output file.
- **Statistics with CTRL-C ( -yc )** — When you use this startup parameter, press CTRL-C rather than using the SHOW-STATS statement to control when to collect statistics during execution.
- **Segment Statistics ( -yd )** — Use this startup parameter to write segment statistics, such as the number of segments and their total size, to the output file.
- **Statistics with Cross-reference ( -yx )** — Use this startup parameter to collect procedure call statistics.

The following example shows the -y startup parameter statistics generated on execution of execlog.p. (See Run-time diagnostics on page 178 for sample code of execlog.p.)

### Sample -y startup parameter statistics

```
Fri Aug 22 09:19:19 2003

Progress Startup Parameters:

-pf C:\Progress\OpenEdge\startup.pf,-cpinternal ISO8859-1,-cpstream
ISO8859-1,-cpcoll Basic,-cpcase Basic,-d mdy,-numsep 44,-numdec 46,
(end .pf),-db sports,-1,-y

Progress client startup options:

 -A  = 0       | -d  = mdy     | -D  = 100     | -h  = 7
 -hs = 0       | -Mpte = 0     | -Mxs= -1      | -nb = 90      | -Nb =0
 -NL = 0       | -o  = printer| -p  = (null)
 -q  = 0       | -Q  = 0       | -s  = 40      | -T =
 -t  = 1       | -TB = 8       | -TM = 8       | -yy = 1950    | -25 = 0
DataServer startup options:

 -nojoinbysqldb =     0
 -SQLlogging =        0
 -noSQLbyserver =     0
 -noindexhint =       0

Database: sports

 -1  = 1       | -a  = (null)
 -B  = 100     | -c  = 20      | -cl = 0       | -cp = (null)
 -cs = 8       | -da = 0       | -db sports
 -dt =  (null)| -g  = (null)
 -H  =  (null)| -i  = 1       | -I  = 25     | -L  = 8192    | -ld = sports
 -Ma = 0       | -Mf = 0      | -Mi = 1       | -Mm = 1024    | -Mn = 5
 -Mr = 1012    | -n  = 1       | -N  =         | -O  = -1      | -R  = 1
 -RO = 0       | -S  = (null)

Execution buffer map:               09:19:32
    Size          Name
    ----          ----
    2250          adecomm/_pwwinst.r
```

```
    9798          adecomm/_pwexit.r
    3232          adecomm/_kvlist.r
    1867          adecomm/_uniqfil.r
    1837          adecomm/_wfrun.r
    8639          adecomm/_toollic.r
   34355          adecomm/_runcode.r
    2202          adecomm/_osprefx.r
    7589          adecomm/_mnkvals.r
     982          adecomm/_osfext.r
    7033          _login.r
    1496          adecomm/_locard.r
    3249          adecomm/_valpnam.r
   99081          adeuib/_uibinfo.r
   11991          adecomm/_adeload.r
    6992          adecomm/_adehelp.r
    1330          execlog2.p
    3431          C:\Solen\100a\p73584_execlog.ped
   10846          adecomm/_getfile.r
   12425          _prostar.r
    4637         _edit.r
   17850          adeedit/_dlggetf.r
    1476          adecomm/_tmpfile.r
  311125          adeedit/_proedit.r
    4255          adecomm/_ossfnam.r
    1506          adecomm/_locdwb.r
     778          adecomm/_setcurs.r
    3727          adecomm/_adeevnt.r
Program access statistics:    Times    Bytes
Reads from temp file:             0        0
Writes to temp file:              0        0
Loads of .r programs:            26   571218
Saves of compilation .r's:        0        0
Compilations of .p's:             2     4761
Checks of files with stat:      107        0
Memory usage summary:       Current    Max Used       Limit (Bytes)
Stack usage (-s):                60       40944       40960
Local buffer usage:            1952       30016
R-code Execution Buffer:     514984      514984     3170304

Segment Descriptors Usage: (numbers)
 Max Used:      288        Limit:      480
------------------------------------------------
```

**Note:** The database statistics appear because the startup parameter connects to a database (`-db sports , -1`).

## Sample r-code segment statistics

The following example shows the `-yd` r-code segment statistics generated on execution of execlog.p. (See Run-time diagnostics on page 178 for sample code of `execlog.p`.)

```
Per procedure temp file access statistics:
      Segment     Read/Write  Times       Bytes
      -------     ----------  -----       -----
execlog.p
execlog2.p
Per procedure segment information
---------------------------------
      File  Segment   #Segments Total-Size
      ----  -------   --------- ----------
      execlog.p
       Initial       1       424
       A-Code: 1      1       836
```

```
       E-Code: 1      1       160
       Debugger       1       180
       Int-Proc: 1    1       256
       Frame: 1       1       460
       Frame: 2       1       400
       Frame: 3       1       400
       Text: 1        1       175

 execlog2.p
       Initial        1       280
       A-Code: 1      1       320
       E-Code: 1      1       144
       Debugger       1       88
       Frame: 1       1       400
       Text: 1        1       38
-------------------------------------------------
```

### Sample procedure call statistics

The following example shows the `-yx` statistics generated on execution of execlog.p. (See Run-time diagnostics on page 178 for sample code of `execlog.p`.)

```
Thu Jan 03 15:21:40 2002
Procedure call statistics:  15:21:42
Caller      Callee    Load Size   Calls  Rd Bytes    Reread      Time
<top>     execlog        3291       1         0         0      1953
execlog     execlog2       1270       1         0         0        16
```

For detailed information on statistics startup parameters, see *Startup Command and Parameter Reference*. For a description of the `SHOW-STATS` statement, see *ABL Reference*.

## Collecting temp-table usage statistics

The statistics for temp-tables (available though `_TableStat` and `_IndexStat` virtual system tables) are lost once the temp-tables (and its indexes) are deleted. Therefore, there is a need to collect the statistics for usage of temp-tables. There are two ways to save the table and index statistics for temp-tables. It can be done through the log entry type `TTStats` or by archiving the statistics with the `TableStatHistory/IndexStatHistory` temp-tables. The data from these tables can be written to an XML file before the session ends.

The new logging entry type parameter `TTStats` enables logging of temp-table usage statistics in the corresponding client or server log files.

If you enable `TTStats` with a logging level of 2 or greater, and the startup parameter `-tttablerangesize` is set to a value greater than 0, the ABL virtual machine (AVM) logs an entry that contains temp-table statistics and includes the table name, and the procedure or class name. The AVM writes the data to the log file before the temp-table gets deleted.

If you enable the `TTStats` parameter with a logging level of 3 or greater, and the startup parameter `-ttindexrangesize` is set to a value greater than 0, the AVM logs an entry that contains index statistics and includes the index name, and the procedure or class name. The AVM writes the data to the log file before the index of the temp-table gets deleted.

For more information on these startup parameters, see *Startup Command and Parameter Reference*.

### Log entry for temp-table statistics

The following example displays the log entry for temp-table statistics.

```
Table: ttCust (myproc.p)
Create: 10 Update: 5 Delete: 2 Read: 1 OS-Read: 2
```

### Log entry for temp-table index statistics

The following example displays the log entry for temp-table index statistics.

```
Index: ttCust:.Name (myproc.p)
Create: 15 Delete: 5 Read: 3 OS-Read: 4 Split: 1 Block-Delete: 1
```

## Supported virtual system tables for temp-tables

The OpenEdge database engine provides virtual system tables (VSTs), which allow applications to get status and activity information on a given database. ABL provides access to these VSTs. Temp-tables are stored in an OpenEdge database. Therefore, VSTs can support temp-tables. This support allows an application to gather information on a given temp-table, and obtain the number of records that are created, modified, and deleted. It can also examine the status of a database and monitor its performance.

The supported virtual system tables for monitoring deployed clients are listed below:

- _ActBuffer
- _ActIndex
- _ActIOFile
- _ActIOType
- _ActOther
- _ActRecord
- _ActSpace
- _ActSummary
- _Block
- _BuffStatus
- _DbStatus
- _Filelist
- _IndexStat
- _MstrBlk
- _StatBase
- _TableStat
- _Trans
- _UserIndexStat

- _UserIO
- _UserTableStat

**Note:** The `_TableStat` and `_UserTableStat` tables contain exactly the same records, since there is only one user accessing the temp-tables. The same applies to the `_IndexStat` and `_UserIndexStat` temp-tables.

For more information on each of these VSTs, see *Manage the OpenEdge Database*.

## ABL classes for accessing temp-tables

Use the `Progress.Database.VSTTableId` class to access VST temp-tables. This class provides a set of static properties. Each property helps in identifying the VST, when the application needs to access a temp-table VST at runtime. The temp-table ID is used to access the relevant records in a given VST.

For more information on each of these static properties and the `Progress.Database.VSTTableId` class, see *OpenEdge Development: ABL Reference*.

Use the `Progress.Database.TempTableinfo` class to archive temp-table data. It provides methods for getting the handle of a given temp-table VST and also for getting the information of the temp-table by position and by ID.

For more information on each of these static properties and the `Progress.Database.TempTableinfo` class, see *OpenEdge Development: ABL Reference*

## Index Statistics file

The following example shows how to read the `_IndexStat` VST and get the index and table names. You will not see any records if the session is not started with a value greater than 0 for `-ttindexrangesize`, or if there are no temp-tables with an ID in the specified range.

### Read a _IndexStat VST

```
USING Progress.Database.*.
def var hVST as handle.
def var hbVST as handle.
def var hq as handle.
DEF VAR cProc AS CHAR .
DEF VAR cIndex AS CHAR .
hVST = TempTableInfo:GetVSTHandle (VSTTableId:IndexStatId) .
hbVST = hvst:default-buffer-handle.
create query hq.
hq:set-buffers(hbVST).
hq:query-prepare("for each " + hbVST:name).
hq:query-open().
/* for each _IndexStat record */
repeat:
   hq:get-next().
   /* check if there are no more records */
   if hq:query-off-end then leave.
      /* check if index-id represents an index that is currently in scope at this time.

         Calling the overloaded version that returns just the index name and
         the procedure that instantiated the temp-table.
      */
      IF TempTableInfo:GetIndexInfoByID(hbVST::_IndexStat-Id,
OUTPUT cIndex,OUTPUT cProc)
THEN DO:
DISP cIndex + " IN " + cProc FORMAT "X(70)"WITH NO-LABELS.
/* disp the data from the _indexStat record */
DISP "Create:" hbVST::_IndexStat-Create
               "Delete:" hbVST::_IndexStat-Delete
               "Block Delete:" hbVST::_IndexStat-BlockDelete
               "Read:" hbVST::_IndexStat-Read
               "OSRead:" hbVST::_IndexStat-OSRead
               "Split:" hbVST::_IndexStat-Split.
         PAUSE.
   CLEAR ALL.
           END.
END.
```

## Table Statistics file

The following table shows how to read the `_TableStat` VST. You will not see any records if the session is not started with a value greater than 0 for `-tttablerangesize`, or if there are no temp-tables with an ID in the specified range.

### Read a _TableStat table

```
USING Progress.Database.*.
def var hVST as handle.
def var hbVST as handle.
def var hq as handle.
DEF VAR cTableName AS CHAR.
DEF VAR cProc AS CHAR .
hVST = TempTableInfo:GetVSTHandle(VSTTableId:TableStatId).
hbVST = hvst:default-buffer-handle.
create query hq.
hq:set-buffers(hbVST).
hq:query-prepare("for each " + hbVST:name).
hq:query-open().
/* for each _TableStaT record */
repeat:
   hq:get-next().
   /* check if there are no more records */
   if hq:query-off-end then leave.
      /* check if the id represents a temp-table in scope at this
         time. Calling the overloaded version that returns the temp-table
         name and the procedure name that instantiated the temp-table.
      */
      IF TempTableInfo:GetTableInfoByID(hbVST::_tableStat-Id,
                                        OUTPUT cTableName, OUTPUT cProc) THEN DO:
       /* display some info about the temp-table and the data from the
         _TableStat table
*/
       DISP cTableName FORMAT "X(20)"
            cProc FORMAT "X(20)" NO-LABEL
            "Create:" hbVST::_TableStat-Create
            "Update:" hbVST::_TableStat-Update
            "Delete:" hbVST::_TableStat-Delete
            "Read:"   hbVST::_TableStat-Read
            "OSRead:" hbVST::_TableStat-OSRead.
       PAUSE.
     END.
END.
```

## Progress.Database.TempTableInfo:GetTableStatHistoryHandle()

In order to analyze the data of application temp-tables that are deleted during the session, archive the statistical information provided by `_TableStat` and `_IndexStat` virtual system tables. If you set the `Progress.Database.TempTableInfo: ArchiveTableStatistics` to true, the AVM will archive and maintain the information in a temp-table. You cannot delete the temp-table when archiving is enabled.

The temp-table name is in the table archive, and the index name is in the index archive. The procedure used for creating the table is also archived. The application is able to look at the statistics for a particular temp-table on a given procedure for each instantiation of the temp-table individually, or accumulate the values to obtain the overall statistics for a particular temp-table across all instantiations in a session. Thus, `_TableStat` and `_IndexStat` provide control over how you want to handle the temp-tables. The `ArchiveTableStatistics` property and `ArchiveIndexStatistics` property enables the archiving of statistics for a temp-table and its index. These properties belong to `Progress.Database.TempTableinfo`

---

**Note:** The statistics are maintained only if their IDs are within the range specified by these startup parameters:`-ttbasetable`, `-ttbaseindex`, `-tttablerangesize`, and `-ttindexrangesize`.

---

The following example shows how to archive the statistics for temp-tables and indexes.

## Archiving temp-tables and indexes

```
USING Progress.Database.*.
OUTPUT TO Arch.out.
DEFINE VARIABLE hTableArch AS HANDLE      NO-UNDO.
DEFINE VARIABLE hbTableArch AS HANDLE     NO-UNDO.
DEFINE VARIABLE hIndexArch AS HANDLE      NO-UNDO.
DEFINE VARIABLE hbIndexArch AS HANDLE     NO-UNDO.
DEFINE VARIABLE i AS INTEGER    NO-UNDO INITIAL 1.
DEFINE VARIABLE n AS INTEGER    NO-UNDO INITIAL 1.
DEFINE VARIABLE retOK AS LOGICAL     NO-UNDO.
DEFINE VARIABLE tth AS HANDLE      NO-UNDO.
CREATE TEMP-TABLE tth.
DEFINE VARIABLE tth2 AS HANDLE      NO-UNDO.
CREATE TEMP-TABLE tth2.
DEFINE VARIABLE hq AS HANDLE      NO-UNDO.
DEFINE VARIABLE hHdl AS HANDLE      NO-UNDO.
tth:CREATE-LIKE(TEMP-TABLE ttDept:HANDLE).
tth:TEMP-TABLE-PREPARE("tth").
tth2:CREATE-LIKE(TEMP-TABLE ttEmp:HANDLE).
tth2:TEMP-TABLE-PREPARE("tth2").
Progress.Database.TempTableInfo:ArchiveTableStatistics = YES.
Progress.Database.TempTableInfo:ArchiveIndexStatistics = YES.
DELETE OBJECT tth.
DELETE OBJECT tth2.
hTableArch = Progress.Database.TempTableInfo:GetTableStatHistoryHandle().
hIndexArch = Progress.Database.TempTableInfo:GetIndexStatHistoryHandle().
IF VALID-HANDLE(hTableArch) THEN
    hbTableArch = hTableArch:DEFAULT-BUFFER-HANDLE.
IF VALID-HANDLE(hIndexArch) THEN
    hbIndexArch = hIndexArch:DEFAULT-BUFFER-HANDLE.
create query hq.
hHdl = hbTableArch.
run PrintArchRecs.
hHdl = hbIndexArch.
run PrintArchRecs.
DELETE OBJECT hq.

procedure PrintArchRecs:
    MESSAGE SKIP hHdl:NAME.
    hq:SET-BUFFERS(hHdl).
    hq:QUERY-PREPARE("preselect each " + hHdl:NAME ).
    hq:QUERY-OPEN().
    REPEAT:
       hq:GET-NEXT().
       MESSAGE "   ".
       if hq:QUERY-OFF-END then leave.
       IF hHdl:NAME = "_TableStatHist" THEN
           MESSAGE hHdl::_Table-name hHdl::_Prog-name hHdl::_Delete-Timestamp.
       ELSE
           MESSAGE hHdl::_Index-name hHdl::_Prog-name hHdl::_Delete-Timestamp.
    END.  /* repeat  */
end PROCEDURE.
```

The following example displays the output file for archived temp-tables and indexes.

## Output file for archived temp-tables and indexes

```
_TableStatHist
tth Archive.p 08/05/2011 15:30:56.433
tth2 Archive.p 08/05/2011 15:30:56.433
_IndexStatHist
tth.idx1 Archive.p 08/05/2011 15:30:56.433
tth2.idx2 Archive.p 08/05/2011 15:30:56.433
```

## Analyzing ABL code

Use the following resources and tools, in addition to the OpenEdge Debugger, to analyze your ABL code:


- Debug Alert startup parameter on page 187
- Enhanced logging features on page 187
- STARTUP-PARAMETERS attribute of the SESSION system handle on page 187
- ProSpy Plus on page 188


### Debug Alert startup parameter

The Debug Alert (`-debugalert`) startup parameter places a Help button on alert dialog boxes displayed by a client application that provides access to a Stack Trace dialog box and the Debugger. For more information, see Debugging from alert dialog boxes on page 42.

### Enhanced logging features

Use the enhanced logging features available in OpenEdge to monitor and control all aspects of logging, including the extent and type of information logged, and the size and retention of log files. For more information, see Logging in OpenEdge on page 99.

### STARTUP-PARAMETERS attribute of the SESSION system handle

Use the `STARTUP-PARAMETERS` attribute of the `SESSION` system handle to obtain a comma-separated list of all startup parameters specified for the current OpenEdge session. The character string returned includes startup parameters from the following potential sources:

- Default startup parameter file (`$DLC/startup.pf`)
- File specified by the `$PROSTARTUP` environment variable
- Startup parameters specified on the command line
- Startup parameters specified in a parameter file (`.pf`)

The following example shows the `STARTUP-PARAMETERS` attribute output generated by executing execlog.p. (See also Run-time diagnostics on page 178.)

### Sample STARTUP-PARAMETERS output

```
-pf C:\Progress\OpenEdge\startup.pf,-cpinternal ISO8859-1,-cpstream
ISO8859-1,-cpcoll Basic,-cpcase Basic,-d mdy,-numsep 44,-numdec 46,
(end .pf),-db sports,-1
```

See *ABL Reference* for detailed information on the `STARTUP-PARAMETERS` attribute.

### ProSpy Plus

ProSpy Plus is a debugging tool you can use to record a stack trace of ABL execution at run time. It also records the instantiation of classes (generated by the NEW statement) and the invocation of methods defined within classes (generated by the SUPER and DELETE OBJECT statements or the INVOKE() method). You can turn tracing on and off at the beginning and end of execution, with the click of a button. The results of tracing, which include all procedures, triggers, publish, subscribe, and unsubscribe executions, as well as user defined functions executed during the tracing period, appear in a hierarchical treeview interface. This allows a user to navigate through a stack trace to see as much or as little, execution information as they need or want.

To learn more about this tool, consult the online help for ProSpy Plus.

## Generating a protrace file for a running program

You can force the creation of a `protrace` file, which contains stack traces and other information, for an OpenEdge process that is currently executing. OpenEdge automatically writes this file when a program terminates abnormally, but in some cases you might want to generate it while the program is running. For more information about the file, see protrace file on page 176.

You use the script `proGetStack.bat` (Windows) or `proGetStack` (UNIX) for this purpose. To run the script, enter the following at a command prompt:

```
proGetStack process_ID
```

Generating a `protrace` file in this way does not stop execution of the OpenEdge session.

**Note:** On Windows, the `protrace` file generated by the `proGetStack.bat` script does not include an AVM stack trace.

# Analyzing procedures and queries

Use the following resources and tools to analyze your procedures and queries:

- `XREF` option of the `COMPILE` statement
- `INDEX-INFORMATION` attribute of the query object handle

## COMPILE XREF option

The `XREF` option of the `COMPILE` statement provides a variety of static data about an ABL procedure, including:

- Other procedures that are `RUN`
- Text strings that exist in the procedure
- Code pages, include files, and shared variables used
- Events published, subscribed, and unsubscribed to
- Database tables accessed
- Indexes used for a particular database query

The following example shows the XREF option output generated by executing execlog.p. (See Run-time diagnostics on page 178 for sample code of execlog.p.)

### Sample XREF option output

```
execlog.p execlog.p 1 COMPILE execlog.p
execlog.p execlog.p 1 CPINTERNAL iso8859-1
execlog.p execlog.p 1 CPSTREAM ibm850
execlog.p execlog.p 1 STRING "ix" 2 NONE UNTRANSLATABLE
execlog.p execlog.p 3 STRING "Customer" 8 NONE UNTRANSLATABLE
execlog.p execlog.p 3 ACCESS sports.Customer Name
execlog.p execlog.p 3 SEARCH sports.Customer Name WHOLE-INDEX
execlog.p execlog.p 4 ACCESS sports.Customer Name
execlog.p execlog.p 4 STRING "x(20)" 5 NONE TRANSLATABLE  FORMAT
execlog.p execlog.p 9 STRING "->,>>>,>>9" 10 NONE TRANSLATABLE  FORMAT
execlog.p execlog.p 14 STRING "ii" 2 NONE UNTRANSLATABLE
execlog.p execlog.p 15 STRING "Number of Customers is " 23 NONE TRANSLATABLE
execlog.p execlog.p 15 STRING "x(23)" 5 NONE TRANSLATABLE  FORMAT
execlog.p execlog.p 15 STRING "->,>>>,>>9" 10 NONE TRANSLATABLE  FORMAT
execlog.p execlog.p 16 STRING "ii" 2 LEFT TRANSLATABLE
execlog.p execlog.p 16 STRING "                          ----------------" 42 NONE
UNTRANSLATABLE
execlog.p execlog.p 16 PROCEDURE internalA,,INPUT ii INTEGER
execlog.p execlog.p 16 STRING "INTERNALA" 9 NONE UNTRANSLATABLE
execlog.p execlog.p 16 STRING "ix" 2 LEFT TRANSLATABLE
execlog.p execlog.p 16 STRING "----------------" 16 NONE UNTRANSLATABLE
execlog.p execlog.p 16 STRING "Name" 4 LEFT TRANSLATABLE
execlog.p execlog.p 16 STRING "Name" 4 NONE UNTRANSLATABLE
execlog.p execlog.p 16 STRING "----------------------" 22 NONE UNTRANSLATABLE
execlog.p execlog.p 16 STRING "Name" 4 NONE UNTRANSLATABLE
execlog.p execlog.p 8 RUN execlog2.p
```

For detailed information on the XREF option, see *ABL Reference*.

## INDEX-INFORMATION attribute of the query object handle

The INDEX-INFORMATION attribute provides data on a dynamic query similar to what XREF provides on a static query. For example, given a dynamic query q1, where the QUERY-PREPARE string is:

```
FOR EACH customer WHERE customer.custnum < 3, EACH order OF customer,
  EACH orderline
```

The query returns these attribute values:

```
q1:INDEX-INFORMATION[1] = "Cust-Num"
q1:INDEX-INFORMATION[1] = "Cust-Order"
q1:INDEX-INFORMATION[1] = "WHOLE-INDEX,order-line"
```

For detailed information on the INDEX-INFORMATION attribute, see *ABL Reference*.

To obtain additional query information and statistics use the QryInfo log entry type. For detailed information see the Query information logging.

# Index

_ (underscore) files *88*
-noevalprops startup parameter *52–53*

4GLMessages log entry type *111*
4GLProvider log entry type *115*
4GLTrace log entry type *111*
4GLTrans log entry type *111*

## A

ABL debugger control
  features *25*
ABL lines
  executable and non-executable *28*
ABL message logging
  log entry types *111*
  log entry types detail *122*
  message format *122*
ABL stack trace pane *50*
ABL trace logging
  log entry types *111*
  log entry types detail *122*
  message format *122*
  return message format *122*
ABL transaction logging
  log entry types *111*
  log entry types detail *124*
  message format *124*
Accelerator keys
  described *56*
Actional log entry type *115*
Active procedures
  defined *28*
Add Watch dialog box *62*
AdminProvider log entry type *115*
AIA logging
  log entry types *112*
  log entry types detail *126*
  message format *126*
AiaDefault log entry type *112*
AiaMgmt log entry type *112*
AiaProp log entry type *112*
AiaRqst log entry type *112*
AiaUBroker log entry type *112*
Analyzing ABL code *187*
Analyzing procedures and queries *188*
Application control *22*
Application mode
  defined *34*
  exiting *34*
Applications
  stacking *39*
AppServer logging
  log entry types *112*
  log entry types detail *127*
  message format *127*
Archiving temp-tables and indexes *185*
ASDefault log entry type *112*
ASPlumbing log entry type *112*
Asynchronous remote procedures
  debugging *30*
Attach to Process dialog box *63*
Attachable mode
  debug ready *36–37*, *63*
  defined *35*
  detaching *46*
  r-code requirements *36*

## B

BLOB data type *51*
Breakpoint indicator
  defined *49*
Breakpoint management
  features *24*
Breakpoints
  adding to procedure *79*, *81*
  adding watchpoint to watch *83*
  conditional *80*
  defined *23*, *64*
  edit *76*
  edit watchpoint *77*
  error-based *81*
  procedure names and line numbers *79*
  setting watchpoints *81*
  watchpoint conditions *83*
  watchpoint expressions *82*
Breakpoints dialog box *64*
brkrLogAppend property *106*
brkrLogEntryTypes property *117*
brkrLoggingLevel property *119*
brkrLogThreshold property *108*
brkrNumLogFiles property *109*
BrokerClient log entry type *115*

## C

Clear Log File (-clearlog) startup parameter *106*
CLEAR-LOG( ) method *106*
-clearlog startup parameter *106*
CLOB data type *51*

CLOSE-LOG( ) method *107*
Code path analysis
features *24*
Command entry
menu options *56*
Commands
proDebugConfig *37*
proDebugEnable *25*
COMPILE XREF option *188*
Conditional breakpoints *80*
Connects log entry type *116*
core dump file *177*
CRC, See Cyclic redundancy check
Current procedure
defined *29*
Cursor log entry type *116*
Cyclic redundancy check *88*

## D

Data
debugger context *28*
Data analysis
features *23*
Data management
features *23*
Data manipulation
features *23*
Data menu *60*
Data types
BLOB *51*
CLOB *51*
DATETIME *51*
LONGCHAR *51*
Database connection logging
log entry types *112*
log entry types detail *128*
message format *128*
DataServer logging
DataServer client context logging *128*
Dataview dialog box
Array tab *68*
Attributes tab *66*
Fields tab *67*
Value tab *69*
DATETIME data type *51*
DB.Connects log entry type *112*
Debug Alert (-debugalert) startup parameter *187*
Debug listing
defined *49*
displaying *49*
name *56*
Debug listing files
creating *86*
deleting *87*
format *49*
Debug listing files *(continued)*
generating
r-code *88*
source code *88*
locating *87*
Debug menu *58*
Debug ready
Attach to Process dialog box *63*
Debugger window *36*, *63*
proDebugConfig command *37*
Ready Attachable Debugger (-debugReady) startup parameter *36*
-debug startup parameter *41*
-debugalert startup parameter *187*
Debugger
features *22–24*
functions *22*
modes of operation *33*
overview *26*
starting from alert boxes *42*
starting from an ABL procedure *45*
starting from the ADE Desktop *40*
starting from the AppBuilder *40*
starting from the command line *41*
starting from the Procedure Editor *39*
starting in attachable mode *46*
starting with the -debug startup parameter *41*
system requirements *21*
Debugger (-debug) startup parameter *41*
Debugger context
active procedures *28*
callback procedures *28*
current procedure *29*
data *28*
defined *27*
execution status *28*
focus *29*
inactive procedures *28*
persistent procedures *28*
procedure call stack *29*
procedures *28*
Debugger dialog boxes
Add Watch *62*
Attach to Process *63*
Breakpoints *64*
Dataview
Array tab *68*
Attributes tab *66*
Field tab *67*
Value tab *69*
Dynamic Object Tracking
Control Monitoring tab *71*
View Objects tab *75*
Edit Breakpoint
At Line tab *76*
On Error tab *76*

Debugger dialog boxes *(continued)*
Edit Breakpoint *(continued)*
Watchpoint tab *76*
Find *77*
New Breakpoint
At Line tab *77*
On Error tab *77*
Watchpoint tab *77*
Open *83*
Preferences
Attachable tab *85*
Font tab *84*
Settings tab *84*
Select Process *86*
Debugger files
attachable mode location *87*
listing files *86*
listing generation *87*
Procedure files *87*
Debugger window
ABL stack trace pane *50*
accelerator keys *56*
Data menu *60*
debug listing *49*
Debug menu *58*
Diagnostics menu *59*
Edit menu *57*
File menu *56*
Help menu *60*
layout *48*
listing indicators *49*
overview *47*
panes *48*
Run to Line command *50*
Search menu *57*
source code pane *49*
status bar *62*
text selection *49*
title bar *56*
tool bar *60*
toolbar *60*
variables pane *50*
View menu *57*
viewing arrays *51*
viewing BLOB data type *51*
viewing CLOB data type *51*
viewing DATETIME data type *51*
viewing LONGCHAR data type *51*
viewing ProDataSet objects *53*
viewing Progress.Languge.ProObject datatype *52*
watches pane *53*
Window menu *60*
Debugger window dialog boxes, See Debugger dialog boxes
Debugger window menu bar, See Menu bar
Debugger window menus, See Menus
Debugger-application interaction
application control *27*
debugger control *27*
Debugger-client interaction *26*
Debugging
Asynchronous remote procedures *30*
asynchronous requests *30*
distributed debugging *30*
enabling *25*
remote debugging *30*
Debugging modes
application mode *34*
attachable mode *46*
stand-alone mode *35*
Debugging sessions
defined *21*
overview *26*
-debugReady startup parameter *36*
Diagnostics menu *59*
Distributed debugging
applications *29*
compatibility issues *30*
definition *30*
environment *29*
remote debugging *30*
DS.Cursor log entry type *112*
DS.Performance log entry type *115*
DSLOG-MANAGER *117*
DSLOG-MANAGER system handle
LOG-ENTRY-TYPES attribute *117*
Logging level attribute *119*
Dump files
core *177*
procore *176*
protrace *176*, *188*
Dynamic object logging
log entry types *113*
log entry types detail *131*
message format *135*
Dynamic object monitoring
features *24*
Dynamic Object Tracking dialog box
Control Monitoring tab *71*
View Objects tab *75*
DynamicApi log entry type *115*
DynObjects.Class log entry type *113*
DynObjects.DB log entry type *113*
DynObjects.Other log entry type *113*
DynObjects.UI log entry type *113*
DynObjects.XML log entry type *113*

## E

Edit Breakpoint dialog box
At Line tab *76*
On Error tab *76*

Edit Breakpoint dialog box *(continued)*
Watchpoint tab *76*
Edit menu *57*
Enabling
debugging *25*
Error-based breakpoints *81*
Event logging
generation *139*
information logged *140*
log entry types *113*
log entry types detail *139*
logged events *139*
message formats *144*
sample logs *148*
stack *139*
Executable line
defined *28*
Execution monitoring *24*
Execution status *28*

## F

Failure conditions *177*
File access logging
log entry types *113*
log entry types detail *149*
message format *149*
File menu *56*
FileID log entry type *113*
Filenames
logging *109*
Files
debug listing *86*
Files, procedure, See Procedure files
Find dialog box *77*

## G

Garbage collection *58*, *61*
Generating query information *154*

## H

Help menu *60*

## I

Ignored operation loggingg
log entry types detail *150*
IgnoredOps log entry type *113*
Inactive procedures
defined *28*
Index statistics file *183*
INDEX-INFORMATION attribute of query object handle *189*
Indicators
breakpoint *49*
line pointer *49*
Interrupt function
defined *23*

## L

Line pointer indicator
defined *49*
Lines
executable and nonexecutable *28*
Listing
indicators *49*
Listing generation *87*
Listing management
features *25*
Listing name *49*, *56*
Log entry types
4GLMessages *111*
4GLProvider *115*
4GLTrace *111*
4GLTrans *111*
Actional *115*
AdminProvider *115*
AiaDefault *112*
AiaMgmt *112*
AiaProp *112*
AiaRqst *112*
AiaUBroker *112*
ASDefault *112*
ASPlumbing *112*
BrokerClient *115*
Connects *116*
Cursor *116*
DB.Connects *112*
DS.Cursor *112*
DS.Performance *115*
DynamicApi *115*
DynObjects.Class *113*
DynObjects.DB *113*
DynObjects.Other *113*
DynObjects.UI *113*
DynObjects.XML *113*
FileID *113*
IgnoredOps *113*
MsgDebug *115*
MsgrTrace *113*
NSPlumbing *113*
Performance *116*
PoolMgmt *115*
ProEvents.UI.Char *113*
ProEvents.UI.Command *113*
ProEvents.UI.Other *113*
Properties *115*
QryInfo *114*

Log entry types *(continued)*
RefCounts *115*
RunProcs *115*
SAX *114*
SessionPool *115*
SOAPProc *115*
SQL *116*
Temp-Tables *114*
Trans *116*
TTStats *115*
UBroker.All *115*
UBroker.AutoTrim *115*
UBroker.Basic *115*
UBroker.ClientFSM *115*
UBroker.ClientMemTrace *115*
UBroker.ClientMsgQueue *115*
UBroker.ClientMsgStream *115*
UBroker.ServerFSM *115*
UBroker.ServerMemTrace *115*
UBroker.ServerMsgQueue *115*
UBroker.ServerMsgStream *115*
UBroker.Stats *115*
UBroker.ThreadPool *115*
WSADefault *115*
WSAObject *115*
WSAObjectPool *115*
WSDLDoc *115*
Log Entry Types (-logentrytypes) startup parameter *117*
Log entry types (Client context) *110*
Log entry types (Server context) *115*
Log files
clearing *106*
closing *107*
rolling over *109*
size threshold *107*
writing user messages *106*
Log Size Threshold (-logthreshold) startup parameter *108*
LOG-ENTRY-TYPES attribute *117*
LOG-MANAGER system handle
CLEAR-LOG( ) method *106*
CLOSE-LOG( ) method *107*
LOG-ENTRY-TYPES attribute *117*
LOG-THRESHOLD attribute *108*
LOGGING-LEVEL attribute *119*
NUM-LOG-FILE attribute *108*
WRITE-MESSAGE( ) method *107*
LOG-MANAGER system handle attributes and methods *100*
LOG-THRESHOLD attribute *108*
logAppend property *106*
logEntryTypes property *117*
-logentrytypes startup parameter *117*
logFile property *109*
logged objects *131*
Logging categories
ABL messages *122*
ABL trace *122*
ABL transaction *124*
AIA *126*
AppServer *127*
Database connection *128*
Dynamic object *131*
Events *139*
File access *149*
Ignored operation *150*
NameServer *151*
Query information *152*
SAX parser *165*
Unified Broker *167*
Web Services Adapter *168*
WebSpeed Messenger *173*
Logging infrastructure *99*
Logging Level (-logginglevel) startup parameter *119*
Logging levels *117*
Logging settings *119*
LOGGING-LEVEL attribute *119*
-logginglevel startup parameter *119*
loggingLevel property *119*
logThreshold property *108*
-logthreshold startup parameter *108*
LONGCHAR data type *51*

## M

Menu accelerator keys *56*
Menus
accelerator keys *56*
data *60*
debug *58*
diagnostics *59*
edit *57*
file *56*
help *60*
search *57*
view *57*
window *60*
Message formats
ABL message logging *122*
ABL trace logging *122*
ABL transaction logging *124*
AIA logging *126*
AppServer logging *127*
Database connection logging *128*
Dynamic object logging *135*
Event logging *144*
File access logging *149*
NameServer logging *151*
Query information logging *162*
SAX parser logging *165*
Unified Broker logging *167*

Message formats *(continued)*
Web Services Adapter logging *173*
WebSpeed Messenger logging *173*
Mnemonics *56*
MsgDebug log entry type *115*
MsgrTrace log entry type *113*

## N

NameServer logging
log entry types *113*
log entry types detail *151*
message formats *151*
New Breakpoint dialog box
At Line tab *77*
On Error tab *77*
Watchpoint tab *77*
No Evaluation of Properties (-noevalprops) startup parameter *52–53*
Nonexecutable line
defined *28*
NSPlumbing log entry type *113*
NUM-LOG-FILES attribute *108*
Number of Log Files (-numlogfiles) startup parameter *109*
numLogFiles property *109*
-numlogfiles startup parameter *109*

## O

Objects
ProDataSet *53*
Progress.Languge.ProObject *52*
tracking *70*
Open and incremental execution *22*
Open dialog box *83*
OpenEdge Management or OpenEdge Explorer setting log characteristics *119*

## P

Panes *48*
Performance log entry type *116*
Persistent procedures
debugger context *28*
PoolMgmt log entry type *115*
Preferences dialog box
Attachable tab *85*
Font tab *84*
Settings tab *84*
Procedure call stack
defined *22*
Procedure execution
features *22*
Procedure files
CRC values *88*
generating source code listings *88*
hidden *88*
r-code listing generation *88*
Procedure interruption features, See Breakpoint management features
Procedure names
leading underscore (_) *29*
Procedure termination
features *23*
Procedures
debugger context *28*
persistent
making active *28*
procore dump file *176*
ProDataSet object *53*
proDebugConfig command *37*
proDebugEnable command *25*
ProEvents.UI.Char log entry type *113*
ProEvents.UI.Command log entry type *113*
ProEvents.UI.Other log entry type *113*
proGetStack script *188*
Progress.Languge.ProObject datatype *52*
Properties
brkrLogAppend *106*
brkrLogEntryTypes *117*
brkrLoggingLevel *119*
brkrLogThreshold *108*
brkrNumLogFiles *109*
disabling evaluation of *52–53*
logAppend *106*
logEntryTypes *117*
logFile *109*
logging settings *119*
loggingLevel *119*
logThreshold *108*
numLogFiles *109*
srvLogWatchDogInterval *110*
srvrLogAppend *106*
srvrLogEntryTypes *117*
srvrLoggingLevel *119*
srvrLogThreshold *108*
srvrNumLogFiles *109*
Properties log entry type *115*
ProSpy Plus *188*
protrace dump file *176*, *188*

## Q

QryInfo log entry type *114*
Query information logging
code constructs *154*
identifying indices *157*
information logged *156*
log entry types *114*

Query information logging *(continued)*
log entry types detail *152*
message formats *162*
query plan *156*
query statistics *157*
sample logs *162*
starting logging *152*

## R

R-code
requirements for attachable debugging mode *36*
usage statistics *179*
Ready Attachable Debugger (-debugReady) startup parameter *36*
RefCounts log entry type *115*
Remote debugging
definition *30*
Rolled-over log files *109*
Run to Line command *50*
Run-time diagnostics *178*
RunProcs log entry type *115*

## S

Sample logs
event logging *148*
first pre-pass query output *162*
query plan output *162*
query statistics output *162*
second pre-pass query output *162*
SAX log entry type *114*
SAX parser logging
log entry types *114*
log entry types detail *165*
message format *165*
sample log *165*
Search menu *57*
Select Process dialog box *86*
Stack trace *42*
SESSION handle
STARTUP-PARAMETERS attribute *187*
Session management *24*
SessionPool log entry type *115*
SOAPProc log entry type *115*
Source code pane
displaying *49*
indicators *49*
label *49*, *56*
text selection *49*
Specifying
log entry types *116*
log filename *105*
log settings with OpenEdge Management or OpenEdge Explorer *119*
log size threshold *108*
Specifying *(continued)*
logging level *118*
retained log files *108*
SQL log entry type *116*
srvLogWatchDogInterval property *110*
srvLogAppend property *106*
srvLogEntryTypes property *117*
srvLoggingLevel property *119*
srvLogThreshold property *108*
srvrNumLogFiles property *109*
Stacking applications *39*
Stand-alone mode
defined *35*
Starting the Debugger
from alert boxes *42*
from an ABL procedure *45*
from the ADE Desktop *40*
from the AppBuilder *40*
from the command line *41*
from the Procedure Editor *39*
in attachable mode *46*
with the -debug startup parameter *41*
Startup parameters
Attachable Debugger (-debugReady) *36*
Clear Log File (-clearlog) *106*
Debug Alert (-debugalert) *187*
Debugger (-debug) *41*
Log Entry Types (-logentrytypes) *117*
Log Size Threshold (-logthreshold) *108*
Logging Level (-logginglevel) *119*
Number of Log Files (-numlogfiles) *109*
STARTUP-PARAMETERS attribute *187*
Statistics *179*
Status bar *62*
System handles
LOG-MANAGER *100*
System requirements *21*

## T

Table statistics file *184*
Temp-table usage statistics *181*
Temp-Tables log entry type *114*
Text selection
source code pane *49*
Title bar *56*
Toolbar *60*
Trans log entry type *116*
Transaction monitoring
features *24*
Trigger blocks
debugger context *28*
TTStats log entry type *115*

## U

UBroker.All log entry type *115*
UBroker.AutoTrim log entry type *115*
UBroker.Basic log entry type *115*
UBroker.ClientFSM log entry type *115*
UBroker.ClientMemTrace log entry type *115*
UBroker.ClientMsgQueue log entry type *115*
UBroker.ClientMsgStream log entry type *115*
UBroker.ServerFSM log entry type *115*
UBroker.ServerMemTrace log entry type *115*
UBroker.ServerMsgQueue log entry type *115*
UBroker.ServerMsgStream log entry type *115*
UBroker.Stats log entry type *115*
UBroker.ThreadPool log entry type *115*
Unified Broker logging
    log entry types *115*
    log entry types detail *167*
    message format *167*
Usage statistics *179*

## V

Variables pane *50*
View menu *57*
VST for temp-tables *181*

## W

Watch
    Add Watch option *53*
Watch *(continued)*
    defined *53*
    drag and drop watches *55*
    supported ABL functions *55*
    valid formats *53*
    watching ABL functions *55*
Watches pane *53*
Watchpoints
    adding watchpoint to watch *83*
    conditions *83*
    defined *64*
    edit *77*
    expressions *82*
    setting *81*
Web Services Adapter logging
    log entry types *115*
    log entry types detail *168*
    message format *173*
WebSpeed Messenger logging
    log entry types *113*
    log entry types detail *173*
    message format *173*
Window layout *48*
Window menu *60*
Window settings
    features *25*
WRITE-MESSAGE( ) method *107*
WSADefault log entry type *115*
WSAObject log entry type *115*
WSAObjectPool log entry type *115*
WSDLDoc log entry type *115*