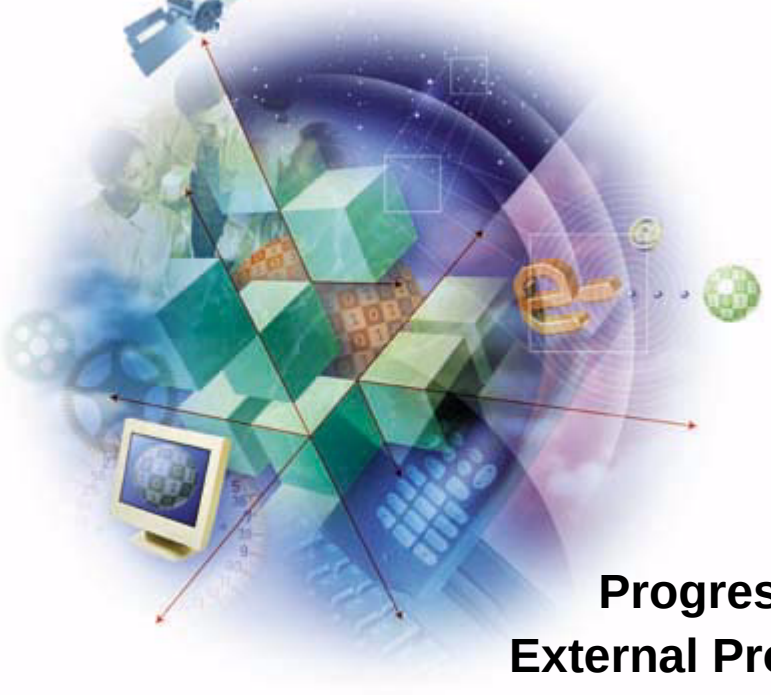


Progress External Program Interfaces

May 2002

Product Code: 4522
Item Number: 89432;V91D

Contents

Figures

Tables

Procedures

Preface

Purpose

Many operating systems and user interfaces provide tools that allow one application to exchange data or use services provided by another application (external program). Progress allows you to use some of these tools as external program interfaces (EPIs) from within a Progress application. This manual describes the EPIs that Progress supports, and explains how to use them from the 4GL to integrate your Progress application with other applications in your operating environment.

Audience

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

Organization Of This Manual

Chapter 3, “System Clipboard”

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

Chapter 4, “Named Pipes”

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

Chapter 5, “Shared Library and DLL Support”

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

Chapter 6, “Windows Dynamic Data Exchange”

Describes how to use dynamic data exchange (DDE) in Windows to send and receive data between your Progress application running as a DDE client and another application running as a DDE server.

Chapter 7, “Using COM Objects In the 4GL”

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

Chapter 8, “ActiveX Automation Support”

Describes Progress support for ActiveX Automation and how to implement a Progress application as an ActiveX Automation Controller from the 4GL.

Chapter 9, “ActiveX Control Support”

Describes Progress support for ActiveX controls in the 4GL, including how to convert an earlier application using VBX controls to the same application using ActiveX controls.

Chapter 10, “Sockets”

Describes Progress support for the use of sockets in the 4GL, including connecting to and disconnecting from a port using sockets and receiving and transmitting data.

Chapter 11, “XML Support”

Describes Progress support for XML in the 4GL, including the receipt and processing of XML documents and the creation and transmission of XML documents.

Chapter 12, “Simple API For XML (SAX)”

Describes Progress support for SAX and provides a SAX API reference.

Chapter 13, “Accessing SonicMQ Messaging From the Progress 4GL”

Introduces the Progress SonicMQ Adapter

Chapter 14, “Using the SonicMQ Adapter”

Tells you how to use the Progress SonicMQ Adapter

Appendix A, “HLC Library Function Reference”

Provides a functional grouping and alphabetical reference of the HLC library functions.

Appendix B, “COM Object Data Type Mapping”

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

Appendix C, “4GL–JMS API Reference”

Provides an API reference for the Progress SonicMQ Adapter

Typographical Conventions

This manual uses the following typographical conventions:

- **Bold typeface** indicates:
  - Commands or characters that the user types
  - That a word carries particular weight or emphasis
- *Italic typeface* indicates:
  - Progress variable information that the user supplies
  - New terms
  - Titles of complete publications
- `Monospaced typeface` indicates:
  - Code examples
  - System output
  - Operating system filenames and pathnames

The following typographical conventions are used to represent keystrokes:

- Small capitals are used for Progress key functions and generic keyboard keys.

  END-ERROR, GET, GO
  ALT, CTRL, SPACEBAR, TAB

- When you have to press a combination of keys, they are joined by a dash. You press and hold down the first key, then press the second key.

  CTRL-X

- When you have to press and release one key, then press another key, the key names are separated with a space.

  ESCAPE H
  ESCAPE CURSOR-LEFT

Syntax Notation

The syntax for each component follows a set of conventions:

- Uppercase words are keywords. Although they are always shown in uppercase, you can use either uppercase or lowercase when using them in a procedure.

  In this example, ACCUM is a keyword:

  SYNTAX

  ```
  ACCUM aggregate expression
  ```

- Italics identify options or arguments that you must supply. These options can be defined as part of the syntax or in a separate syntax identified by the name in italics. In the ACCUM function above, the *aggregate* and *expression* options are defined with the syntax for the ACCUM function in the *Progress Language Reference*.

- You must end all statements (except for DO, FOR, FUNCTION, PROCEDURE, and REPEAT) with a period. DO, FOR, FUNCTION, PROCEDURE, and REPEAT statements can end with either a period or a colon, as in this example:

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

- Square brackets (**[ ]**) around an item indicate that the item, or a choice of one of the enclosed items, is optional.

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

  SYNTAX

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

In some instances, square brackets are not a syntax notation, but part of the language.

For example, this syntax for the INITIAL option uses brackets to bound an initial value list for an array variable definition. In these cases, normal text brackets ([ ]) are used:

SYNTAX

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

NOTE: The ellipsis (**...**) indicates repetition, as shown in a following description.

- Braces (**{ }**) around an item indicate that the item, or a choice of one of the enclosed items, is required.

  In this example, you must specify the items BY and *expression* and can optionally specify the item DESCENDING, in that order:

  SYNTAX

  ```
  { BY expression [ DESCENDING ] }
  ```

  In some cases, braces are not a syntax notation, but part of the language.

  For example, a called external procedure must use braces when referencing arguments passed by a calling procedure. In these cases, normal text braces ({ }) are used:

  SYNTAX

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

- A vertical bar (**|**) indicates a choice.

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

  SYNTAX

  ```
  PRESELECT [ EACH | FIRST | LAST ] record-phrase
  ```

In this example, you must select one of *logical-name* or *alias*:

SYNTAX

```
CONNECTED ( { logical-name | alias } )
```

- Ellipses (**. . .**) indicate that you can choose one or more of the preceding items. If a group of items is enclosed in braces and followed by ellipses, you must choose one or more of those items. If a group of items is enclosed in brackets and followed by ellipses, you can optionally choose one or more of those items.

  In this example, you must include two expressions, but you can optionally include more. Note that each subsequent expression must be preceded by a comma:

  SYNTAX

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

  In this example, you must specify MESSAGE, then at least one of *expression* or SKIP, but any additional number of *expression* or SKIP 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" ] ... }
  ```

- In some examples, the syntax is too long to place in one horizontal row. In such cases, **optional** items appear individually bracketed in multiple rows in order, left-to-right and top-to-bottom. This order generally applies, unless otherwise specified. **Required** items also appear on multiple rows in the required order, left-to-right and top-to-bottom. In cases where grouping and order might otherwise be ambiguous, braced (required) or bracketed (optional) groups clarify the groupings.

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

  SYNTAX

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

  In this example, ASSIGN requires one of two choices: either one or more of *field*, or one of *record*. Other options available with either *field* or *record* are grouped with braces and brackets. The open and close braces indicate the required order of options:

  SYNTAX

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

Example Procedures

This manual provides numerous example procedures that illustrate syntax and concepts. Examples use the following conventions:

- They appear in boxes with borders.
- If they are available online, the name of the procedure appears above the left corner of the box and starts with a prefix associated with the manual that references it, as follows:
  - `e-` — *Progress External Program Interfaces*, for example, `e-ddeex1.p`
  - `lt-` — *Progress Language Tutorials*, for example, `lt-05-s3.p`
  - `p-` — *Progress Programming Handbook*, for example, `p-br01.p`
  - `r-` — *Progress Language Reference*, for example, `r-dynbut.p`

  If the name does not start with a listed prefix, the procedure is not available online.
- If they are not available online, they compile as shown, but might not execute for lack of completeness.

Accessing Files In Procedure Libraries

Documentation examples are stored in procedure libraries, `prodoc.pl` and `prohelp.pl`, in the `src` directory where Progress is installed.

You **must** first create all subdirectories required by a library before attempting to extract files from the library. You can see what directories and subdirectories a library needs by using the PROLIB -list command to view the contents of the library. See the *Progress Client Deployment Guide* for more details on the PROLIB utility.

Creating a Listing Of the Procedure Libraries

Creating a listing of the source files from a procedure library involves running PROENV to set up your Progress environment, and running PROLIB.

1 ♦ From the Control Panel or the Progress Program Group, double-click the Proenv icon.

2 ♦ The Proenv Window appears, with the proenv prompt.

Running Proenv sets the DLC environment variable to the directory where you installed Progress (by default, `C:\Program Files\Progress`). Proenv also adds the DLC environment variable to your PATH environment variable and adds the bin directory (`PATH=%DLC%;%DLC%\bin;%PATH%`).

3 ♦ Enter the following command at the proenv prompt to create the text file `prodoc.txt` which contains the file listing for the prodoc.pl library.

```
PROLIB %DLC%\src\prodoc.pl -list > prodoc.txt
```

Extracting Source Files From Procedure Libraries On Windows Platforms

Extracting source files from a procedure library involves running PROENV to set up your Progress environment, creating the directory structure for the files you want to extract, and running PROLIB.

1 ♦ From the Control Panel or the Progress Program Group, double-click the Proenv icon.

2 ♦ The Proenv Window appears, with the proenv prompt.

Running Proenv sets the DLC environment variable to the directory where you installed Progress (by default, `C:\Program Files\Progress`). Proenv also adds the DLC environment variable to your PATH environment variable and adds the bin directory (`PATH=%DLC%;%DLC%\bin;%PATH%`).

3 ♦ Enter the following command at the proenv prompt to create the `prodoc` directory in your Progress working directory (by default, `C:\Progress\Wrk`):

```
MKDIR prodoc
```

4 ♦ Create the `langref` directory under `prodoc`:

```
MKDIR prodoc\langref
```

5 ♦ To extract all examples in a procedure library directory, run the PROLIB utility. Note that you must use double quotes because “Program Files” contains an embedded space:

```
PROLIB "%DLC%\src\prodoc.pl" -extract prodoc\langref\*.*
```

PROLIB extracts all examples into `prodoc\langref`.

To extract one example, run PROLIB and specify the file that you want to extract as it is stored in the procedure library:

```
PROLIB "%DLC%\src\prodoc.pl" -extract prodoc/langref/r-syshlp.p
```

PROLIB extracts `r-syshlp.p` into `prodoc\langref`.

Extracting Source Files From Procedure Libraries On UNIX Platforms

To extract `p-wrk1.p` from `prodoc.pl`, a procedure library, follow these steps at the UNIX system prompt:

1 ♦ Run the PROENV utility:

```
install-dir/dlc/bin/proenv
```

Running proenv sets the DLC environment variable to the directory where you installed Progress (by default, `/usr/dlc`). The proenv utility also adds the bin directory under the DLC environment variable to your PATH environment variable (`PATH=$DLC/bin:$PATH`).

2 ♦ At the proenv prompt, create the `prodoc` directory in your Progress working directory:

```
mkdir prodoc
```

3 ♦ Create the `proghand` directory under `prodoc`:

```
mkdir prodoc/proghand
```

4 ♦ To extract all examples in a procedure library directory, run the PROLIB utility:

```
prolib $DLC/src/prodoc.pl -extract prodoc/proghand/*.*
```

PROLIB extracts all examples into `prodoc/proghand`.

To extract one example, run PROLIB and specify the file that you want to extract as it is stored in the procedure library:

```
prolib $DLC/src/prodoc.pl -extract prodoc/proghand/p-wrk-1.p
```

PROLIB extracts `p-wrk-1.p` into `prodoc/proghand`.

Progress Messages

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

- Execution messages inform you of errors encountered while Progress is running a procedure (for example, if Progress cannot find a record with a specified index field value).
- Compile messages inform you of errors found while Progress is reading and analyzing a procedure prior to 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 Progress is getting ready to execute (for example, if you entered an invalid startup parameter).

After displaying a message, Progress 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 following execution messages.
- Returns to the Progress Procedure Editor so that you can correct an error in a procedure. This is the usual action taken following compiler messages.
- Halts processing of a procedure and returns immediately to the Procedure Editor. This does not happen often.
- Terminates the current session.

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

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

On the Windows platform, use Progress online help to get more information about Progress messages. Many Progress tools include the following Help menu options to provide information about messages:

- Choose Help→Recent Messages to display detailed descriptions of the most recent Progress message and all other messages returned in the current session.
- Choose Help→Messages, then enter the message number to display a description of any Progress message. (If you encounter an error that terminates Progress, make a note of the message number before restarting.)
- In the Procedure Editor, press the **HELP** key (**F2** or **CTRL-W**).

On the UNIX platform, use the Progress PRO command to start a single-user mode character Progress client session and view a brief description of a message by providing its number. Follow these steps:

1 ♦ Start the Progress Procedure Editor:

```
install-dir/dlc/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.

Other Useful Documentation

This section lists Progress Software Corporation documentation that you might find useful. Unless otherwise specified, these manuals support both Windows and Character platforms and are provided in electronic documentation format on CD-ROM.

Getting Started

Progress Electronic Documentation Installation and Configuration Guide (Hard copy only)

A booklet that describes how to install the Progress EDOC viewer and collection on UNIX and Windows.

Progress Installation and Configuration Guide Version 9 for UNIX

A manual that describes how to install and set up Progress Version 9.1 for the UNIX operating system.

Progress Installation and Configuration Guide Version 9 for Windows

A manual that describes how to install and set up Progress Version 9.1 for all supported Windows and Citrix MetaFrame operating systems.

Progress Version 9 Product Update Bulletin

A bulletin that provides a list of new and changed features by release, a list and description of changes to documentation by release, and critical information about product changes that might require changes to existing code and configurations.

This bulletin also provides information about where to go for detailed information about the new and changed features and documentation.

Progress Application Development Environment — Getting Started (Windows only)

A practical guide to graphical application development within the Progress Application Development Environment (ADE). This guide includes an overview of the ADE and its tools, an overview of Progress SmartObject technology, and tutorials and exercises that help you better understand SmartObject technology and how to use the ADE to develop applications.

Progress Language Tutorial for Windows and *Progress Language Tutorial for Character*

Platform-specific tutorials designed for new Progress users. The tutorials use a step-by-step approach to explore the Progress application development environment using the 4GL.

Progress Master Glossary for Windows and *Progress Master Glossary for Character* (EDOC only)

Platform-specific master glossaries for the Progress documentation set. These books are in electronic format only.

Progress Master Index and Glossary for Windows and *Progress Master Index and Glossary for Character* (Hard copy only)

Platform-specific master indexes and glossaries for the Progress hard-copy documentation set.

Progress Startup Command and Parameter Reference

A reference manual that describes the Progress startup and shutdown commands that you use at the command line, and the startup parameters that you use for Progress processes. This guide also provides information about parameter usage and parameter files.

Welcome to Progress (Hard copy only)

A booklet that explains how Progress software and media are packaged. An icon-based map groups the documentation by functionality, providing an overall view of the documentation set. *Welcome to Progress* also provides descriptions of the various services Progress Software Corporation offers.

Development Tools

Progress ADM 2 Guide

A guide to using the Application Development Model, Version 2 (ADM 2) application architecture to develop Progress applications. It includes instructions for building and using Progress SmartObjects.

Progress ADM 2 Reference

A reference for the Application Development Model, Version 2 (ADM 2) application. It includes descriptions of ADM 2 functions and procedures.

Progress AppBuilder Developer's Guide (Windows only)

A programmer's guide to using the Progress AppBuilder visual layout editor. AppBuilder is a Rapid Application Development (RAD) tool that can significantly reduce the time and effort required to create Progress applications.

Progress Basic Database Tools (Character only; information for Windows is in online help)

A guide for the Progress Database Administration tools, such as the Data Dictionary.

Progress Basic Development Tools (Character only; information for Windows is in online help)

A guide for the Progress development toolset, including the Progress Procedure Editor and the Application Compiler.

Progress Debugger Guide

A guide for the Progress Application Debugger. The Debugger helps you trace and correct programming errors by allowing you to monitor and modify procedure execution as it happens.

Progress Help Development Guide (Windows only)

A guide that describes how to develop and integrate an online help system for a Progress application.

Progress Translation Manager Guide (Windows only)

A guide that describes how to use the Progress Translation Manager tool to manage the entire process of translating the text phrases in Progress applications.

Progress Visual Translator Guide (Windows only)

A guide that describes how to use the Progress Visual Translator tool to translate text phrases from procedures into one or more spoken languages.

Reporting Tools

Progress Report Builder Deployment Guide (Windows only)

An administration and development guide for generating Report Builder reports using the Progress Report Engine.

Progress Report Builder Tutorial (Windows only)

A tutorial that provides step-by-step instructions for creating eight sample Report Builder reports.

Progress Report Builder User's Guide (Windows only)

A guide for generating reports with the Progress Report Builder.

Progress Results Administration and Development Guide (Windows only)

A guide for system administrators that describes how to set up and maintain the Results product in a graphical environment. This guide also describes how to program, customize, and package Results with your own products. In addition, it describes how to convert character-based Results applications to graphical Results applications.

Progress Results User's Guide for Windows and *Progress Results User's Guide for Unix*

Platform-specific guides for users with little or no programming experience that explain how to query, report, and update information with Results. Each guide also helps advanced users and application developers customize and integrate Results into their own applications.

4GL

Building Distributed Applications Using the Progress AppServer

A guide that provides comprehensive information about building and implementing distributed applications using the Progress AppServer. Topics include basic product information and terminology, design options and issues, setup and maintenance considerations, 4GL programming details, and remote debugging.

Progress Internationalization Guide

A guide to developing Progress applications for markets worldwide. The guide covers both internationalization—writing an application so that it adapts readily to different locales (languages, cultures, or regions)—and localization—adapting an application to different locales.

Progress Language Reference

A three-volume reference set that contains extensive descriptions and examples for each statement, phrase, function, operator, widget, attribute, method, and event in the Progress language.

Progress on the Web

A manual that describes how to use the new WebClient, AppServer Internet Adapter, SmartObjects, and SonicMQ Adapter to create applications tailored for Internet, intranet, and extranet environments.

Progress Programming Handbook

A two-volume handbook that details advanced Progress programming techniques.

Database

Progress Database Design Guide

A guide that uses a sample database and the Progress Data Dictionary to illustrate the fundamental principles of relational database design. Topics include relationships, normalization, indexing, and database triggers.

Progress Database Administration Guide and Reference

This guide describes Progress database administration concepts and procedures. The procedures allow you to create and maintain your Progress databases and manage their performance.

DataServers

Progress DataServer Guides

These guides describe how to use the DataServers to access non-Progress databases. They provide instructions for building the DataServer modules, a discussion of programming considerations, and a tutorial.

Each DataServer has its own guide as follows:

- *Progress/400 Product Guide*
- *Progress DataServer for Microsoft SQL Server Guide*
- *Progress DataServer for ODBC Guide*
- *Progress DataServer for ORACLE Guide*

MERANT ODBC Branded Driver Reference

The Enterprise DataServer for ODBC includes MERANT ODBC drivers for all the supported data sources. For configuration information, see the MERANT documentation, which is available as a PDF file in *installation-path*\odbc. To read this file you must have the Adobe Acrobat Reader Version installed on your system. If you do not have the Adobe Acrobat Reader, you can download it from the Adobe Web site at: http://www.adobe.com/products/acrobat/readstep.html.

SQL-89/Open Access

Progress Embedded SQL-89 Guide and Reference

A guide to Progress Embedded SQL-89 for C, including step-by-step instructions on building ESQL-89 applications and reference information on all Embedded SQL-89 Preprocessor statements and supporting function calls. This guide also describes the relationship between ESQL-89 and the ANSI standards upon which it is based.

Progress Open Client Developer's Guide

A guide that describes how to write, build, and deploy Java and ActiveX applications, and Java applets that run as clients of the Progress AppServer. This guide includes information about how to expose the AppServer as a set of Java classes or as an ActiveX server, and how to choose an Open Client distribution package for run time.

Progress SQL-89 Guide and Reference

A user guide and reference for programmers who use interactive Progress/SQL-89. It includes information on all supported SQL-89 statements, SQL-89 Data Manipulation Language components, SQL-89 Data Definition Language components, and supported Progress functions.

SQL-92

Progress Embedded SQL-92 Guide and Reference

A guide to Progress Embedded SQL-92 for C, including step-by-step instructions for building ESQL-92 applications and reference information about all Embedded SQL-92 Preprocessor statements and supporting function calls. This guide also describes the relationship between ESQL-92 and the ANSI standards upon which it is based.

Progress JDBC Driver Guide

A guide to the Java Database Connectivity (JDBC) interface and the Progress SQL-92 JDBC driver. It describes how to set up and use the driver and details the driver's support for the JDBC interface.

Progress ODBC Driver Guide

A guide to the ODBC interface and the Progress SQL-92 ODBC driver. It describes how to set up and use the driver and details the driver's support for the ODBC interface.

Progress SQL-92 Guide and Reference

A user guide and reference for programmers who use Progress SQL-92. It includes information on all supported SQL-92 statements, SQL-92 Data Manipulation Language components, SQL-92 Data Definition Language components, and Progress functions. The guide describes how to use the Progress SQL-92 Java classes and how to create and use Java stored procedures and triggers.

Deployment

Progress Client Deployment Guide

A guide that describes the client deployment process and application administration concepts and procedures.

Progress Developer's Toolkit

A guide to using the Developer's Toolkit. This guide describes the advantages and disadvantages of different strategies for deploying Progress applications and explains how you can use the Toolkit to deploy applications with your selected strategy.

Progress Portability Guide

A guide that explains how to use the Progress toolset to build applications that are portable across all supported operating systems, user interfaces, and databases, following the Progress programming model.

WebSpeed

Getting Started with WebSpeed

Provides an introduction to the WebSpeed Workshop tools for creating Web applications. It introduces you to all the components of the WebSpeed Workshop and takes you through the process of creating your own Intranet application.

WebSpeed Installation and Configuration Guide

Provides instructions for installing WebSpeed on Windows and UNIX systems. It also discusses designing WebSpeed environments, configuring WebSpeed Brokers, WebSpeed Agents, and the NameServer, and connecting to a variety of data sources.

WebSpeed Developer's Guide

Provides a complete overview of WebSpeed and the guidance necessary to develop and deploy WebSpeed applications on the Web.

WebSpeed Product Update Bulletin

A booklet that provides a brief description of each new feature of the release. The booklet also explains where to find more detailed information in the documentation set about each new feature.

Welcome to WebSpeed (Hard copy only)

A booklet that explains how WebSpeed software and media are packaged. *Welcome to WebSpeed!* also provides descriptions of the various services Progress Software Corporation offers.

Reference

Pocket Progress (Hard copy only)

A reference that lets you quickly look up information about the Progress language or programming environment.

Pocket WebSpeed (Hard copy only)

A reference that lets you quickly look up information about the SpeedScript language or the WebSpeed programming environment.

1

Introduction

This chapter introduces the Progress *external program interfaces* (EPIs). These consist of 4GL statements and supporting software that enable a Progress 4GL application to exchange data and services with external non-Progress applications.

To use these interfaces, you must be proficient in the programming tools, applications, and environments you want to access from Progress.

This chapter contains the following sections:

1.1 Using MEMPTR To Reference External Data

The Progress MEMPTR data type provides a means to access (*unmarshal* and *marshal*) data that you plan to pass to and from other application and system environments using the following EPIs:

- UNIX shared object libraries and Windows dynamic link libraries (DLLs)
- Sockets
- XML

A MEMPTR variable references a region of memory set to a size that you specify. Once you declare and initialize the MEMPTR variable to the specified size, you can read and modify the memory region it references as follows:

- Read and write data values to specified locations within the MEMPTR region. These values can contain different numbers of bytes, depending on a data type that you specify for the value read or written.
- Read and write individual bits within a Progress INTEGER variable, whose value you can also read and write to a MEMPTR region.
- Assign values between two MEMPTR variables and between a MEMPTR variable and a RAW variable (or field).
- Indicate the byte order of data you have written so that a target system can interpret the MEMPTR data appropriately. This is primarily useful for exchanging MEMPTR data between different system environments using sockets.

Together, these features allow you to store and access data aggregates (*structures*), including complete Progress database records.

The following sections describe the general capabilities of MEMPTR data, including:

- Comparing MEMPTR and RAW data types
- Initializing and uninitializing MEMPTR variables
- Reading and writing data in MEMPTR variables
- Retrieving and storing pointers
- Setting byte order

For more information on using MEMPTR for a specific EPI, see:

- **Shared libraries** — The "UNIX Shared Library and Windows DLL Support" section in this chapter, then Chapter 5, "Shared Library and DLL Support"
- **Sockets** — The "Sockets" section in this chapter, then Chapter 10, "Sockets"
- **XML** — The "XML" section in this chapter, then Chapter 11, "XML Support"

1.1.1 Comparing MEMPTR and RAW Data Types

Progress provides two data types to store data in a raw (binary) form, MEMPTR and RAW. MEMPTR and RAW data types provide similar features, but serve different purposes, as follows:

- You can use both MEMPTR and RAW variables to store binary data that originates with any other data type in the 4GL or that you import from another environment using an EPI.
- You can use MEMPTR variables to directly exchange data between the 4GL and a supported EPI environment; you can use RAW variables only to exchange data with another 4GL data type, including MEMPTR.
- You can define RAW fields in a Progress database, but you cannot define MEMPTR fields. You can move database fields and records to and from MEMPTR variables using RAW variables as intermediate storage.

In sum, the RAW data type supports the storage and movement of binary data within the local 4GL and Progress database environment, while the MEMPTR data type supports the storage and movement of data between the local 4GL and external 4GL or non-4GL environments.

You can also assign values between MEMPTR and RAW variables directly using the 4GL assignment operator (=). If the target variable is RAW, Progress resizes the target, if necessary, to make it the same size as the source. For both assignment between MEMPTR and RAW variables and assignment between two variables of the same data type, the source and the target variables maintain separate and complete copies of the data.

NOTE: In previous versions of Progress, assignment from one MEMPTR variable to another created a copy of the pointer, not the data. Thus, both variables referred to the same data after the assignment. With Version 9.1, each variable refers to a completely separate copy of the data. However, passing a MEMPTR parameter to a local procedure remains pass by reference, and MEMPTR parameters to a remote procedure are passed by value, as with assignment statements.

1.1.2 Initializing and Uninitializing MEMPTR Variables

After declaring a MEMPTR variable, you must initialize it before you can use it. In general, to initialize a MEMPTR variable, you use the SET-SIZE statement to allocate a region of memory and associate it with the variable.

NOTE: You can also allocate the memory region for a MEMPTR variable using a shared library routine. In this case, you must still use the SET-SIZE statement to initialize the size of the returned memory region in Progress. For more information, see Chapter 5, "Shared Library and DLL Support."

Initializing MEMPTR Variables Using the SET-SIZE Statement

This is the syntax for the SET-SIZE statement:

SYNTAX

```
SET-SIZE ( mptr-name ) = expression
```

The value *mptr-name* is the name of a MEMPTR variable, and *expression* is an integer expression specifying the size, in bytes, of a memory region associated with *mptr-name*. The value of *expression* can also be zero (0), which frees any memory previously allocated to *mptr-name*. Progress uses this size to perform bounds checking that ensures you do not read or write to portions of memory outside of the specified region.

If *mptr-name* is uninitialized (that is, defined but not associated with any memory region) and *expression* is greater than zero (0), the SET-SIZE statement allocates a memory region whose size is specified by *expression*, and associates it with *mptr-name*. If *mptr-name* is already initialized and *expression* is greater than zero (0), the SET-SIZE statement has no effect.

To resize memory allocated to a MEMPTR variable, you must invoke SET-SIZE with an *expression* of zero (0), then invoke SET-SIZE with an *expression* equal to the new allocation.

Checking a MEMPTR Variable For Initialization

You might not be sure whether a particular MEMPTR variable is already initialized when you try to initialize it. You can verify that the variable is initialized and obtain the size of its memory region using the GET-SIZE function. This is the syntax for the GET-SIZE function:

SYNTAX

```
GET-SIZE ( mptr-name )
```

If *mptr-name* is initialized, the GET-SIZE function returns the size of its memory region. Otherwise, it returns zero (0).

NOTE: To return a memory size greater than zero (0), the variable must be fully initialized with the SET-SIZE statement, not just allocated by a shared library routine. For more information, see Chapter 5, "Shared Library and DLL Support."

Freeing Memory Associated With a MEMPTR Variable

The region of memory associated with a MEMPTR variable remains allocated until it is freed. Progress does not automatically free the memory for you. It is up to you to ensure that the memory is freed.

You can free the memory using SET-SIZE with an *expression* of zero (0). SET-SIZE then deallocates (frees) any memory region associated with *mptr-name*. This makes *mptr-name* uninitialized.

NOTE: When working with shared library routines, you must fully understand the memory management provided by these routines before using MEMPTR variables with them effectively. For more information, see Chapter 5, "Shared Library and DLL Support."

1.1.3 Reading and Writing Data

Once you have initialized a MEMPTR variable, you can build a data aggregate (structure) or access an existing structure in the associated memory region using several memory-writing statements and memory-reading functions. Memory-writing statements write values to specified locations in the memory region. Memory-reading functions return values from the specified locations in the memory region. Through an appropriate choice of these statements and functions you can thus copy Progress data types and bit fields to and from MEMPTR variables. You can also copy complete Progress database records to and from MEMPTR variables.

NOTE: Before setting or getting values in a MEMPTR variable, you might want to check the MEMPTR size using the GET-SIZE function. For more information, see the "Initializing and Uninitializing MEMPTR Variables" section.

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

SYNTAX

```
PUT-datatype ( mptr-name , byte-position ) = expression
```

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

SYNTAX

```
GET-datatype ( mptr-name , byte-position )
```

Each PUT-*datatype* statement writes a value (*expression*) of a certain data type to the memory region associated with the MEMPTR variable *mptr-name* at the specified *byte-position*. Each GET-*datatype* function reads and returns the value of a data type from the memory region associated with the MEMPTR variable *mptr-name* at the specified *byte-position*. The *byte-position* in these statements and functions is specified by an integer expression that starts at one (1).

The PUT-STRING statement and GET-STRING function each allow an additional optional parameter that specifies the number of bytes to write or read in the MEMPTR variable. The GET BYTES function has the same parameter, but it is required for this function.

NOTE: The *mptr-name* parameter in these statements and functions can reference RAW as well as MEMPTR variables. However, the EPIs that require MEMPTR variables cannot use RAW variables directly. You must convert RAW values to MEMPTR before using them with these EPIs. You can do this using direct assignment between RAW and MEMPTR variables or by using statements and functions such as PUT-BYTES, GET-BYTES, or GET-RAW.

Memory-writing Statements

Progress provides the PUT-*datatype* statements shown in Table 1–1.

Table 1–1: PUT-*datatype* Statements *(1 of 2)*

Statement	Description
PUT-BYTE	Writes an integer value to the specified 1-byte location.
PUT-SHORT	Writes an integer value to the specified 2-byte location.
PUT-UNSIGNED-SHORT	Writes an unsigned integer value to the specified 2-byte location.
PUT-LONG	Writes an integer value to the specified 4-byte location.

Table 1–1: PUT-*datatype* Statements *(2 of 2)*

Statement	Description
PUT-FLOAT	Writes a decimal value to the specified 4-byte location as a single-precision floating-point value.
PUT-DOUBLE	Writes a decimal value to the specified 8-byte location as a double-precision floating-point value.
PUT-STRING	Writes a character string value to the specified location, either null terminated or for a specified number of bytes.
PUT-BYTES	Writes the contents of a RAW or MEMPTR variable to the specified byte location of a RAW or MEMPTR variable.

For more information on these statements, see the *Progress Language Reference*.

Memory-reading Functions

Progress provides the GET-*datatype* functions shown in Table 1–2.

Table 1–2: GET-*datatype* Functions *(1 of 2)*

Function	Description
GET-BYTE	Returns the integer value of the specified 1-byte location.
GET-SHORT	Returns the integer value of the specified 2-byte location.
GET-UNSIGNED-SHORT	Returns the value of the specified 2-byte location, interpreted as an unsigned integer.
GET-LONG	Returns the integer value of the specified 4-byte location.
GET-FLOAT	Returns the decimal value of the specified 4-byte location, interpreted as a single-precision floating-point value.

Table 1–2: GET-*datatype* Functions *(2 of 2)*

Function	Description
GET-DOUBLE	Returns the decimal value of the specified 8-byte location, interpreted as a double-precision floating-point value.
GET-STRING	Returns the character string value from the specified location, either null terminated or for a specified number of bytes that can include nulls.
GET-BYTES	Returns, as a MEMPTR or RAW value, the specified number of bytes from the specified byte location of a RAW or MEMPTR variable.

For more information on these functions, see the *Progress Language Reference*.

Copying Between Basic Progress Data Types and MEMPTR

Table 1–3 lists the basic Progress data types and how you can copy them in and out of a MEMPTR variable.

Table 1–3: Copying Between Basic Progress Data Types and MEMPTR *(1 of 2)*

Data Type	Copying To/From MEMPTR
DATE	To copy into a MEMPTR: • Use INTEGER(*date-expression*) to obtain an integer value. • Use the PUT-LONG statement to copy integer value to MEMPTR. To copy from a MEMPTR: • Use the GET-LONG function to return the integer value. • Use the DATE(*integer-expression*) function to return the date.
DECIMAL[1]	To copy into a MEMPTR: • Use the PUT-DOUBLE or PUT-FLOAT statement. To copy from a MEMPTR: • Use the GET-DOUBLE or GET-FLOAT function.

Table 1–3: Copying Between Basic Progress Data Types and MEMPTR *(2 of 2)*

Data Type	Copying To/From MEMPTR
INTEGER[1]	To copy into a MEMPTR: • Use the PUT-LONG, PUT-SHORT, PUT-UNSIGNED-SHORT, or PUT-BYTE statements. To copy from a MEMPTR: • Use the GET-LONG, GET-SHORT, GET-UNSIGNED-SHORT, or GET-BYTE functions.
LOGICAL[1]	To copy into a MEMPTR: • Use the PUT-LONG, PUT-SHORT, PUT-UNSIGNED-SHORT, or PUT-BYTE statements. To copy from a MEMPTR: • Use the GET-LONG, GET-SHORT, GET-UNSIGNED-SHORT, or GET-BYTE functions.
RAW	To copy into a MEMPTR: • Assign the RAW value directly using an assignment statement or use the PUT-RAW statement. To copy from a MEMPTR: • Assign the MEMPTR value directly using an assignment statement or use the GET-RAW function.

[1] The choice of exact statement or function to use depends on the data type used by the shared library routine or the socket application with your application is communicating.

Manipulating Bit Values

You can copy bit values up to the size of a Progress INTEGER from one INTEGER value to another. The statement to copy bit values to an INTEGER variable, PUT-BITS, has the following syntax:

SYNTAX

```
PUT-BITS( destination , start-position , count ) = integer-expression
```

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

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

SYNTAX

```
GET-BITS( source , start-position , count )
```

This function returns the INTEGER that represents the value of the number of bits specified by *count* starting at the specified low-order bit position (*start-position*) within the INTEGER variable specified by *source*.

Thus, you can store bit values in MEMPTR variables and return bit values from MEMPTR variables by using the PUT-LONG statement and GET-LONG function to store and return the corresponding INTEGER expression that contains the bit pattern.

Copying Between Database Records and MEMPTR

Copying a database record to and from a MEMPTR variable relies on the RAW-TRANSFER statement. This statement allows you to copy a whole database record buffer to a RAW variable or to copy a RAW variable to a database record buffer.

Thus, to store a database record in a MEMPTR variable:

1 ♦ Copy the record buffer to a RAW variable using the RAW-TRANSFER statement.

2 ♦ Assign the RAW variable to the MEMPTR variable or to the specified position in the MEMPTR variable using the PUT-BYTES statement.

To retrieve a database record from a MEMPTR variable:

1 ♦ Assign the MEMPTR variable to a RAW variable or use the GET-BYTES function to copy the specified bytes from the MEMPTR to the RAW variable.

2 ♦ Copy the RAW variable to the record buffer using the RAW-TRANSFER statement.

1.1.4 Retrieving and Storing Pointers

In some cases (especially for shared library routines), you might have to obtain a pointer to the memory region associated with a MEMPTR variable. You might need this value, for example, to build a structure that contains a pointer to another structure. You can get a pointer to a MEMPTR region by using the GET-POINTER-VALUE function. This is the syntax for the GET-POINTER-VALUE function:

SYNTAX

```
GET-POINTER-VALUE ( memptr-name )
```

For example, to retrieve the pointer to a memory region specified by a MEMPTR variable (BitMapInfo) and store it in the first byte position of another MEMPTR variable (BitMaps), you can use the following statement (assuming 32-bit pointers):

```
PUT-LONG(BitMaps,1) = GET-POINTER-VALUE(BitMapInfo).
```

You can also store the address of a memory item into a MEMPTR variable by using the SET-POINTER-VALUE statement. Here is the syntax:

SYNTAX

```
SET-POINTER-VALUE ( memptr-name ) = address
```

For example, to store an address located at the second byte of a MEMPTR variable (BitMaps) into another MEMPTR variable (BitMapInfo), you can use the following statement:

```
SET-POINTER-VALUE(BitMapInfo) = GET-LONG(BitMaps,2)
```

For more information on the GET-POINTER-VALUE function and the SET-POINTER-VALUE statement, see the *Progress Language Reference*.

1.1.5 Setting Byte Order

When passing MEMPTR data between system environments (such as can happen using sockets), you must help ensure that the communicating applications agree on the byte order for any data types that are transferred using the MEMPTR variable. Most machines follow one of two schemes for ordering bytes:

- **Little-endian** — Low-order bytes are stored starting at the lowest address location reserved for a data type, with progressively higher-order bytes stored at the higher byte positions. For example, 7 is stored in a four-byte (hex) integer as follows, where addresses increase from left to right:

  ```
  07 00 00 00
  ```

- **Big-endian** — High-order bytes are stored in the lowest address location reserved for a data type with progressively lower-order bytes stored at the higher byte positions. For example, 7 is stored in a four-byte (hex) integer as follows, where addresses increase from left to right:

  ```
  00 00 00 07
  ```

Progress ensures that MEMPTR data is interpreted correctly using MEMPTR write/read statements and functions as long as you indicate what byte order the MEMPTR data that you write uses. You can do this using the SET-BYTE-ORDER statement:

SYNTAX

```
SET-BYTE-ORDER( memptr-name ) = integer-expression
```

Given the name of the MEMPTR variable (*memptr-name*), you can set one of these keyword values for *integer-expression*:

- **HOST-BYTE-ORDER** — A value of one (1) that indicates the byte ordering of the machine where the statement executes.
- **BIG-ENDIAN** — A value of two (2) that indicates big-endian byte ordering.

  NOTE: Internet protocols use big-endian byte ordering.
- **LITTLE-ENDIAN** — A value of three (3) that indicates little-endian byte ordering.

By default, all MEMPTR variables use the byte ordering of the machine where they are defined.

Note that the SET-BYTE-ORDER statement does not change the existing order of bytes in the MEMPTR data. It only indicates how subsequent MEMPTR write/read statements and functions interpret the data. You can also return the current byte order setting using the GET-BYTE-ORDER function. For more information, see the *Progress Language Reference*.

1.2 Host Language Call Interface

The Host Language Call (HLC) Interface is a Progress feature that allows you to write and call your own C language functions directly from a Progress procedure using the 4GL CALL statement. Using HLC library functions, your C functions can read from or write to shared variables and shared buffer fields defined in the Progress context, interact with the Progress display, test for Progress interrupts, and access Progress-managed timer services. Ultimately, you can use HLC to access devices not supported directly in Progress (such as process sensors or ATM terminals), and exchange data between your Progress procedures and these devices.

1.2.1 HLC and Progress

HLC provides access to third-party application program interfaces from your 4GL applications, similar to using shared libraries in the Windows environment (see the "UNIX Shared Library and Windows DLL Support" section).

Figure 1–1 shows a top-down structure diagram for a Progress 4GL application that calls your application functions using HLC.

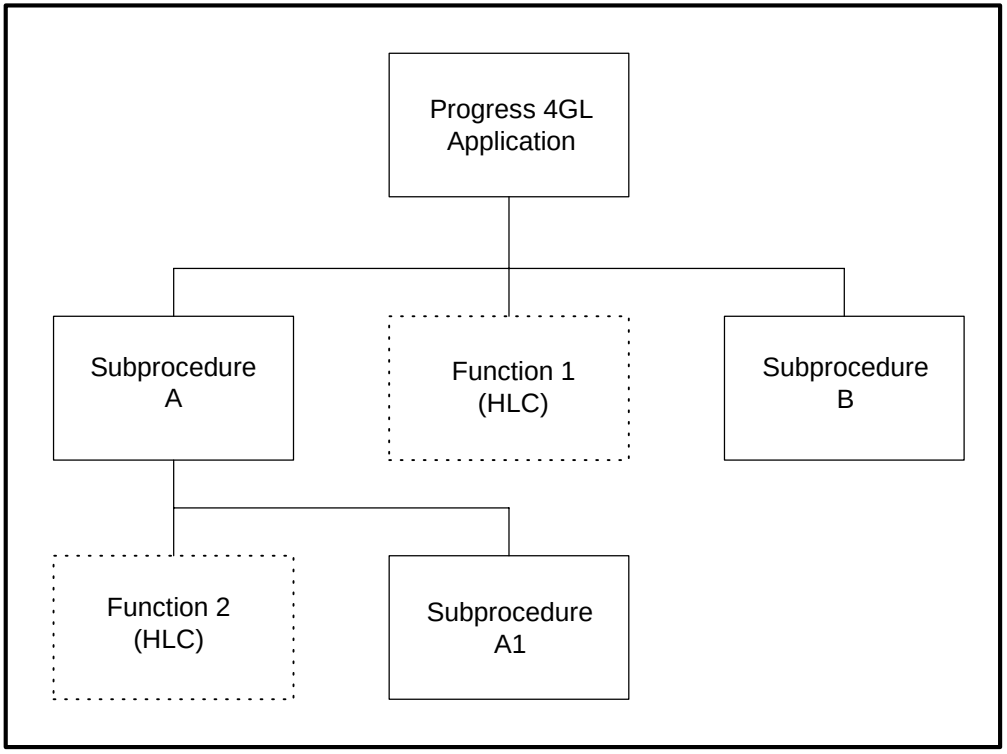


Figure 1–1: Progress Application Calling Functions With HLC

1.2.2 Requirements For Using HLC

To use HLC, you must be proficient with the C language and the C language compiler and linker on your system.

1.3 System Clipboard

The *system clipboard* allows a window system user to transfer data between one widget (or application) and another using cut, copy, and paste operations. Each window system provides its own style of clipboard support. However, it generally allows the user to perform these clipboard operations in exactly the same way between all applications on the system using some combination of the mouse and keyboard.

1.3.1 The System Clipboard and Progress

Figure 1–2 shows the typical interactions between a Progress 4GL application and other graphical applications using the clipboard.

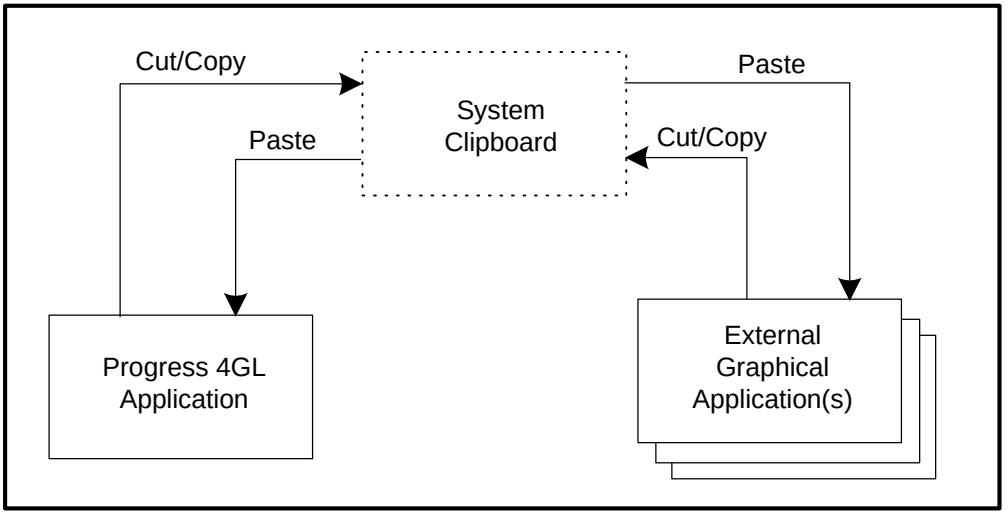


Figure 1–2: Progress Interactions Using the System Clipboard

The dotted box emphasizes that, from the user's viewpoint, the clipboard is essentially invisible.

Progress supports clipboard interactions with other graphical applications in the Windows environments. In character mode, Progress supports clipboard interactions between fields in a single Progress application. Using the CLIPBOARD system handle and an appropriate set of user-interface triggers, you can define the response of your Progress application to the standard clipboard operations on your system. For example, you can eliminate cut operations (that remove data) and provide only copy and paste; or you can specify how the data is transferred, whether all or part of it is transferred, and where it goes when it is pasted into your Progress application.

1.3.2 Requirements For Using the CLIPBOARD Handle

The minimum requirement for using the CLIPBOARD handle is a thorough knowledge of how the clipboard operates on your operating system from the user's viewpoint. You can then use the CLIPBOARD handle to provide or modify that functionality in your Progress applications.

1.4 Named Pipes

Named pipes provide the basic interprocess communications (IPC) mechanism in the UNIX environment. This IPC mechanism allows two processes on UNIX to send data to each other as if they are reading and writing sequential files. Each process opens a separate pipe for input or output to the other, and the input process waits for the output process to send data each time it reads its input pipe. Named pipes are also supported in the Windows NT environment and generally behave similarly to UNIX named pipes.

1.4.1 Named Pipes and Progress

Figure 1–3 shows typical data exchanges between a Progress 4GL application and an external application using named pipes.

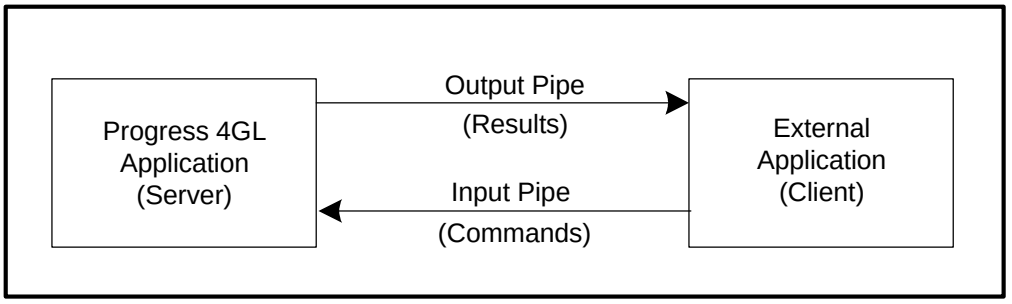


Figure 1–3: Progress Exchanging Data Using Named Pipes

A Progress database client accesses named pipes in the same manner as any input or output file. Typically, the Progress database client acts as a 4GL server, receiving 4GL requests from a non-Progress UNIX or Windows NT application through an input pipe, executing the 4GL statements, and returning the results to the UNIX or Windows NT application through an output pipe. Chapter 4, "Named Pipes" emphasizes this type of client/server interaction with Progress using named pipes.

1.4.2 Requirements For Using Named Pipes

The minimum requirement for working with named pipes in Progress is a working knowledge of the UNIX operating system and its shell commands or of the Windows NT operating system, and of how the non-Progress application you want to communicate with uses pipes. Proficiency in the C programming language is also helpful when working with the named pipe examples in this manual.

1.5 UNIX Shared Library and Windows DLL Support

The UNIX and Windows operating systems support the use of an executable file that contains compiled functions or routines that can be linked to an application at runtime rather than at build time. This executable file is called a *shared object* or *shared library* on UNIX and a *dynamic link library* (DLL) on Windows. This dynamic linking capability promotes a building block approach to application development; third-party software packages provide much of their functionality in shared libraries. It also makes function upgrades more easily available to applications that use them.

Throughout this section and in the corresponding chapter of this book (Chapter 5, "Shared Library and DLL Support"), the term "shared library" is used when a statement is applicable to both the UNIX and Windows systems. "UNIX shared library" is used if a statement is applicable only to UNIX; "DLL" is used for statements applicable only to Windows.

1.5.1 Shared Libraries and Progress

Figure 1–4 shows a top-down structure diagram for a Progress 4GL application that calls shared library functions.

NOTE: Both Progress graphical and character applications can access DLLs in Windows.

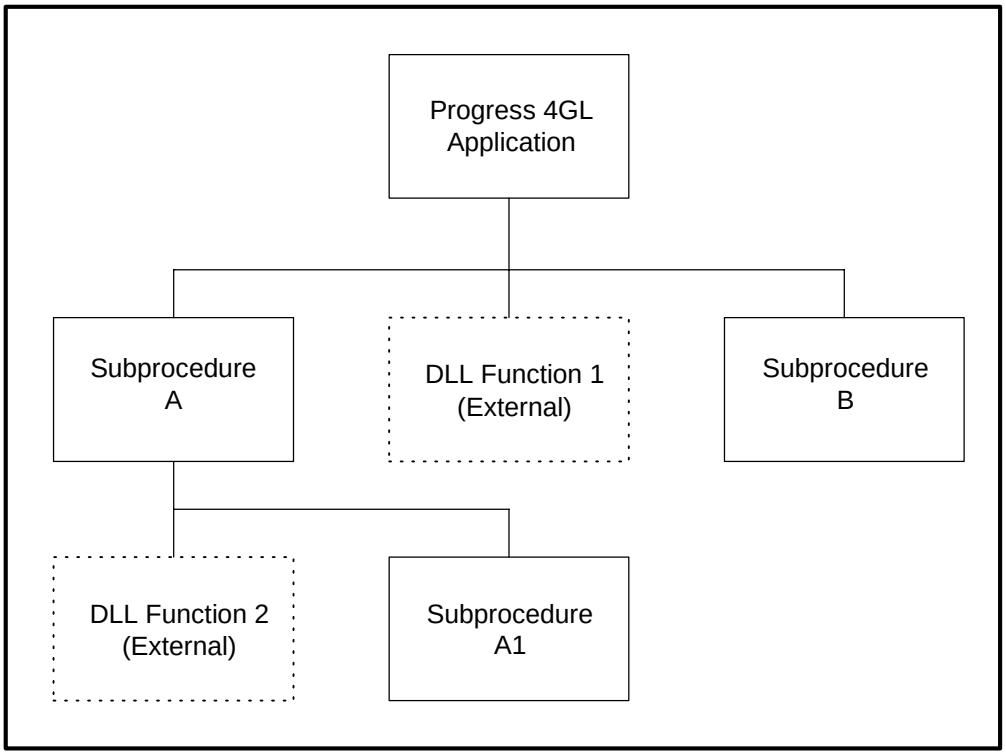


Figure 1–4: Progress Application Calling DLL Functions

The dotted boxes emphasize that although shared library functions are called from within a Progress application, they actually reside in external shared libraries.

A Progress application can access a shared library function much like an internal 4GL procedure. The client declares the function and its file using a PROCEDURE statement, defining all function parameters with DEFINE PARAMETER statements. Progress provides a special set of parameter data types to match C and Windows data types, and a set of statements and functions to build and access C structures from Progress. The client executes the shared library function exactly like a 4GL procedure, using the RUN statement.

1.5.2 Requirements For Using Shared Libraries

The minimum requirement for working with shared libraries in Progress is knowledge of how to use the function parameters. Declaring shared library functions in Progress also requires a basic knowledge of C programming and of the operating system(s) on which your applications run.

It might be helpful for at least one programmer proficient in Windows or UNIX and C to provide the required shared library function declarations in include files. They might also want to write procedures or include files that appropriately create, set, and read any C structures used by the functions. All other Progress programmers only have to use the procedures and include files provided by this programmer. Otherwise, they execute the shared library functions like any 4GL procedure.

1.6 Windows Dynamic Data Exchange

Dynamic data exchange (DDE) is a protocol Windows provides for interprocess communications. Using this protocol, two applications communicate in a client/server relationship in which the client initiates the communications and the server exchanges data with and provides services to the client. This is a very flexible IPC mechanism that enables a range of capabilities from simple data transmission between two applications to the ability for multiple applications to “work” in each other’s environments. For example, a word processing application might create and modify spreadsheets in a spreadsheet application, and the spreadsheet application might, in turn, create and modify documents in the word processing application.

1.6.1 DDE and Progress

Figure 1–5 shows a series of IPC exchanges between a Progress 4GL application and another Windows application using DDE.

NOTE: Both Progress graphical and character applications can use DDE in Windows.

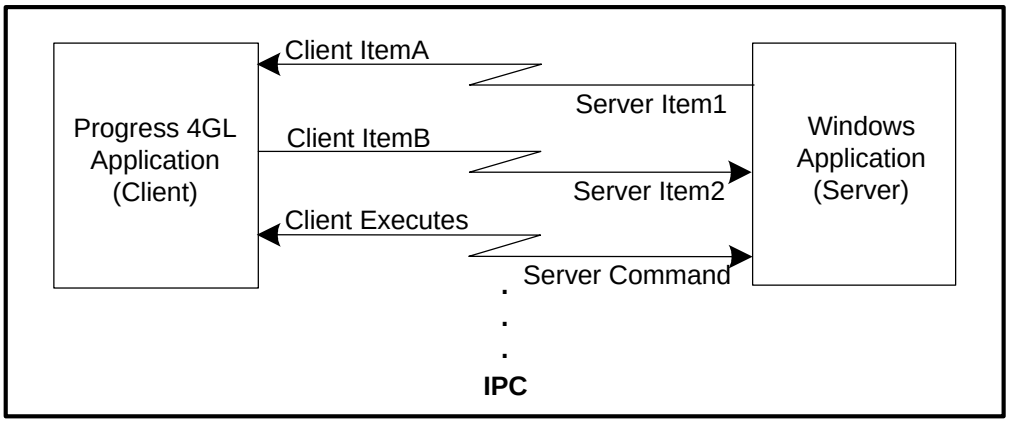


Figure 1–5: Progress Exchanging Data Using DDE

In this example, the Progress application sets the value of Progress ItemA from Server Item1; sets the value of Server Item2 from Progress ItemB; and executes a command on the Server, possibly returning a data value or error condition.

Progress supports DDE as a client only. This allows Progress database clients to communicate with any other Windows application with DDE server capability. Examples of Windows applications with DDE server capability include the Windows Program Manager, Microsoft Excel for Windows, and Visual Basic applications. As a DDE client, Progress can, for example, create and modify worksheets in Excel, and at the same time automatically receive notification of updates to worksheet cells (data items) from Excel. Excel, as the DDE server, provides this notification by sending an event that Progress can handle in a trigger.

1.6.2 Requirements For Using DDE

The minimum requirement for working with DDE in Progress is familiarity with the way your DDE server applications provide data and services to DDE clients. Each server application has its own way of naming data items and groups of data items in its own environment. It also usually provides a set of commands that you can execute with DDE that direct the server to perform server-supported application actions (for example, to create documents or spreadsheets). For information on using DDE with each application, see the application documentation.

1.7 COM Objects: Automation Objects and ActiveX Controls

COM objects are encapsulated Windows application objects that conform to specifications of the Microsoft Component Object Model. As such, COM objects provide functionality for an application that might not otherwise be supported by Progress. This allows you to acquire functional elements for your Progress applications from third-party vendors as well as from Progress Software Corporation.

The COM standard allows Progress to access a COM object through its properties, methods, and events. This is analogous to how Progress provides access to widget attributes, methods, and events. However, Progress provides access to COM objects through an industry-standard mechanism and provides access to widgets through a proprietary mechanism.

1.7.1 COM Objects Supported In Progress

Progress supports two classes of COM objects:

- ActiveX Automation objects
- ActiveX controls

ActiveX Automation Objects

ActiveX Automation objects (or just *Automation objects*) are COM objects that encapsulate all or part of an application in a stand-alone executable (EXE file) or dynamic link library (DLL) file. These Automation objects make the encapsulated functionality available to another application.

Similar to DDE, your Progress application functions in a client/server relationship to the application that provides an Automation object. As such, Progress functions as an ActiveX Automation Controller and accesses the Automation object in the application that functions as an ActiveX Automation Server.

For example, your Progress application reads and writes data values in the Automation Server by accessing properties and methods of an available Automation object, such as a spread sheet or word-processing document. Depending on the server application, you might even be able to create new Automation object instances (new spread sheets or documents) in the Automation Server from within your Progress application.

Figure 1–6 shows a series of IPC exchanges between a Progress 4GL application and the Automation objects of an Automation Server:

NOTE: Both Progress graphical and character applications can use ActiveX Automation in Windows.

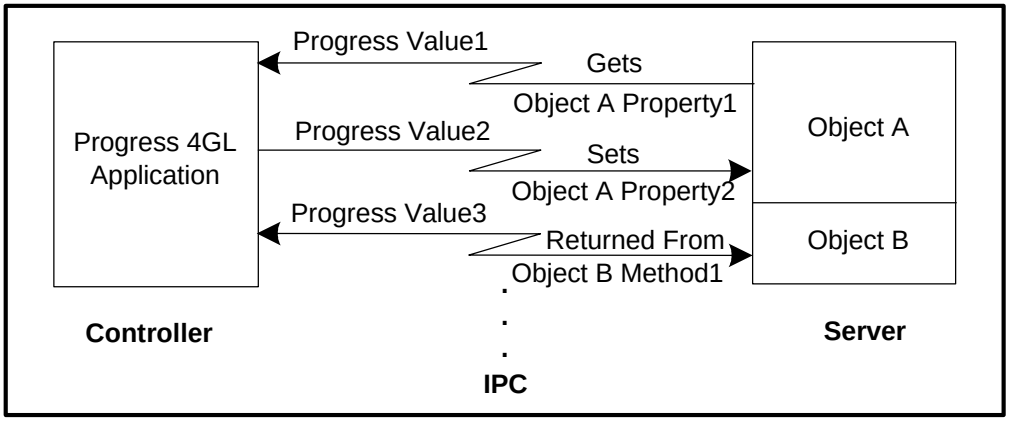


Figure 1–6: Progress Using ActiveX Automation

In this example, the Progress application gets Progress Value1 from Object A Property1; sets Object A Property2 from Progress Value2; and gets Progress Value3 from a call to Method1 of Object B, possibly after creating an instance of Object B.

Compare this functionality with DDE (see Figure 1–5). The effects are very similar. However, the access to properties and methods provided by the component model of ActiveX Automation is much more straightforward, robust, and flexible.

ActiveX Controls

ActiveX controls (OCXs) are COM objects that rely on COM standards, including ActiveX Automation, to communicate with an application and also to provide a mechanism to generate events. ActiveX controls reside only in DLL files that provide the complete implementation, which often includes a user-interface component. As such, ActiveX controls are directly analogous to 4GL widgets, but often include a variety of capabilities not available with the widgets built into the 4GL. For example, you can find ActiveX controls that function as calendars, pie charts, bar graphs, gauges, meters, and even communications, timing, and parsing controls that have no user-interface components.

Whether or not an ActiveX control includes a user-interface component, Progress supports a user-interface widget, the CONTROL-FRAME widget, to make the control available to your application. The control-frame widget anchors the ActiveX control to your application. This widget physically orients the control in the Progress user interface, but provides no other services for accessing it. A separate but related COM object, the control-frame COM object, provides the real control container support. This special Progress-supported COM object contains and provides direct access to the ActiveX control from the 4GL.

NOTE: Unlike with ActiveX Automation, you can use ActiveX controls only in Progress graphical applications.

Support for VBX controls in earlier versions of Progress requires you to use special methods on a control-container widget to access any VBX control property or method. However for ActiveX controls, Progress allows you to access control properties and methods directly, without reference to the control-frame widgets that anchor them.

Figure 1–7 shows how Progress supports ActiveX controls in the 4GL.

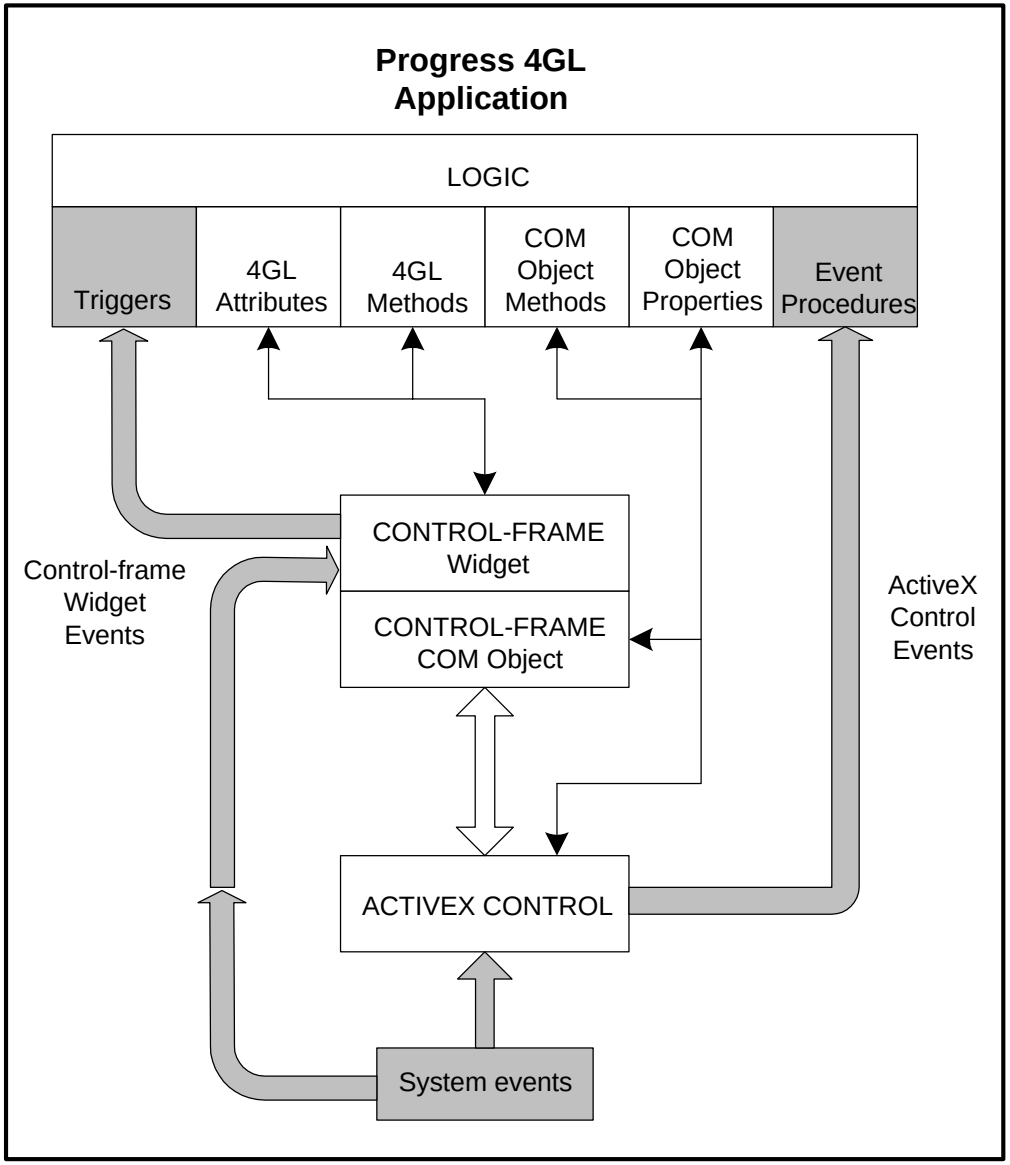


Figure 1–7: Progress Interface To ActiveX Controls

The control-frame widget and COM object work together to connect the ActiveX control to your application. As you might suspect, Progress supports different types of handles for widgets and COM objects. Thus, you access control-frame widgets using widget handles and access all COM objects (including ActiveX controls) using component handles. It is these component handles that allow you to gain direct access to the properties and methods of an ActiveX control (or Automation object).

Similar to VBX control support in earlier versions, Progress allows you to define event procedures to handle events generated directly by ActiveX controls. You can also handle events on the control-frame widget with user-interface triggers. Control-frame widget events (like TAB or LEAVE) allow you to coordinate user-interface actions between ActiveX controls and Progress widgets.

1.7.2 Support For COM Object Properties and Methods

In Progress, the first step to access a COM object is to obtain its component handle. Once you have the component handle, you can use it to access properties and methods supported by the COM object. To support these property and method references, Progress also provides automatic mappings between COM data types and Progress data types. This allows you to pass COM object property, method, and event parameter values directly as Progress data items without the need for data conversion functions.

For more information on accessing COM object properties and methods, see Chapter 7, “Using COM Objects In the 4GL.”

1.7.3 Support For Automation Object Events

Automation objects can generate events in response to an action performed on the object, such as the creation of a Word document or the printing of an Excel Workbook. Progress supports event notification for ActiveX Automation objects using a built-in method on the COM object, ENABLE-EVENTS. Once events are enabled, Progress searches for a running or persistent procedure matching the event name. For more information on handling events for Automation objects, see Chapter 8, “ActiveX Automation Support.”

1.7.4 Support For ActiveX Control Events

ActiveX controls respond to events much like Progress widgets. ActiveX controls that provide a user interface typically generate user-interface events, similar to user-interface widgets. However, ActiveX controls can support other types of functionality that generate other types of events, such as events that notify the arrival of a message for a communications application or the change in temperature of a manufacturing process for a data acquisition application.

Also, unlike widget events, ActiveX control events can pass parameters like a procedure call. Thus, Progress provides a type of internal procedure (*OCX event procedure*) to handle ActiveX control events. You can handle any ActiveX control event using an OCX event procedure.

Progress also allows you to handle certain events on the control-frame widget instead of on the ActiveX control. When ActiveX controls have focus, they generally take over the input and your application receives most events directly as ActiveX control events. However, when an ActiveX control has no equivalent event or when it is necessary to manage the orientation of the control in the Progress user interface, you can handle some input actions as field-level widget events on the control-frame. You can handle these widget events using the standard 4GL ON statement, but Progress executes only one event handler (a Progress trigger or an OCX event procedure) per event.

For more information on handling events for ActiveX controls, see Chapter 9, "ActiveX Control Support."

1.7.5 COM Object Sources

Any number of Automation objects and ActiveX controls might be available on your Windows system, depending on its configuration and the applications you have installed. In general, you can get information on the available COM objects by using the COM Object Viewer that comes installed with Progress. For more information on this viewer, see Chapter 7, "Using COM Objects In the 4GL."

Automation Objects

Since Automation objects are generally part of a stand-alone application that functions as an Automation Server, you must have the application installed that provides the Automation objects that you need for your Progress application. For information on the available Automation objects in a particular Automation Server application, see the documentation for that application or use the Progress COM Object Viewer.

ActiveX Controls

Three ActiveX controls come installed with Progress for Windows. These include a combo box, a spin button, and a timer control. For more information on these controls, see Chapter 9, "ActiveX Control Support."

Visual Basic Professional provides a starter set of ActiveX controls and includes basic documentation on working with ActiveX controls. Other commercial vendors and countless sources of shareware and freeware offer ActiveX control packages of varying quality.

CAUTION: The control vendor is responsible for following COM standards. Any deviation might result in a control that does not work in the Progress environment.

1.7.6 Requirements For Using Automation Objects

The main requirement for using an Automation object is that the Automation Server application must be installed on your system and ready to execute. For more information, see the documentation on ActiveX Automation support that comes with your Automation Server application.

1.7.7 Requirements For Using ActiveX Controls

You can access ActiveX controls in two modes:

- Design mode
- Run mode

Design-mode (or design-time) access allows you to modify properties that initialize the control and define it for use in an application. For many controls, these properties affect such attributes as color and size, but also enable and disable other special features unique to each ActiveX control. In general, design-time properties affect the appearance and initial internal state of an ActiveX control. Design-time properties are generally readable, but might not be writable at run time.

Run-mode (or runtime) access allows your application to interact with the control, responding to events, invoking methods, and getting and setting properties that affect the ActiveX control at run time. Run-time properties might not be readable or writable at design time.

The requirements for working in each mode differ.

Design-mode Requirements

To define an ActiveX control for use in a Progress application, you must:

- Have the ActiveX control installed in your Windows environment — This includes one or more DLL files. Install the ActiveX control according to the vendor's instructions. The primary control file usually has a `.ocx` extension. Note that other files might also be required and installed, as well.

- Have a license installed that allows you to access the ActiveX control in design mode — Not all ActiveX controls require licenses for design-time access, but most commercial ActiveX controls do. Licenses generally come with the vendor's installation, and are either recorded in the registry or in a license file, often with a `.lic` extension. Progress stores the license information for the ActiveX controls that it installs in the registry.

- Use the Progress AppBuilder to create an instance of the ActiveX control in your application — The AppBuilder allows you to select the ActiveX control in design mode and place an instance of it into your application. It also allows you to set the values the properties of the control will initially assume at run time, and saves these values in a separate binary file. This file (by default) has the same name as your application file with the `.wrx` extension. The .wrx file contains the definitions of all ActiveX control instances in the corresponding application (.w) file.

Aside from using the AppBuilder to create ActiveX control instances, you can also code OCX event procedures and control-frame event triggers with minimal effort using the AppBuilder. The AppBuilder event list includes both ActiveX control events and Progress control-frame widget events. The control-frame event names appear first, followed by the ActiveX control event names prefixed by OCX. For more information on accessing this list of events, see the *Progress AppBuilder Developer's Guide*.

Run-mode Requirements

An ActiveX control is always in run mode when you execute a Progress application that includes it. No license is required for run-time access to an ActiveX control. Thus, to deploy and execute an application that contains ActiveX controls, you must provide at least the following files:

- The .w file generated by the AppBuilder for your application, or the compiled r-code file
- The .wrx file saved with your application
- The .ocx file for each ActiveX control contained in your application
- Other DLL support files and files containing data and bitmaps that come installed with the ActiveX control.

The .wrx file contains most of the information required to use each ActiveX control instance at run time, often including references to bitmaps and other external files.

1.7.8 Programming Requirements

For more information on programming COM objects in the 4GL, see Chapter 7, "Using COM Objects In the 4GL." For information specific to Automation objects, see Chapter 8, "ActiveX Automation Support," and for information specific to ActiveX controls, see Chapter 9, "ActiveX Control Support."

NOTE: On non-Windows systems, any 4GL code that references COM objects can compile, but generates run-time errors when executed. Therefore, isolate any COM object references in multi-platform code by using Preprocessor directives.

1.8 Sockets

Sockets are software communication end-points that allow one process to communicate with another on the same machine or across a network. The Progress implementation supports TCP/IP sockets in the 4GL that allow one 4GL application to establish a connection with another 4GL or non-4GL application, and thus to communicate with that application on the same machine or across a network.

1.8.1 Reasons To Use Sockets

Sockets allow your 4GL application to interact with other applications built using any language and deployed on any network machine using a standard communications model. 4GL sockets are integrated with the 4GL event model so you can use a single mechanism to handle user-interface events, AppServer asynchronous request completion events, and socket events. Applications that lend themselves to socket communications with the 4GL include:

- Ticker tape applications for financial markets
- Data acquisition for manufacturing and processes
- Web servers
- Mail servers
- Any other message-based applications

Alternative 4GL mechanisms that allow socket access include the Host Language Call Interface (HLC) and shared library access described in this manual. However, 4GL sockets provide a native 4GL mechanism that is much easier to program and that is well-integrated with the 4GL event model.

The Progress implementation provides low-level access to TCP/IP sockets. In the 4GL, a socket client and server each send and receive data as a stream of bytes, accessed as a MEMPTR data type. The formatting of this data stream is entirely application dependent. You can marshal and unmarshal data streams according to your application requirements using the 4GL statements and functions available to manipulate MEMPTR data (see the "Using MEMPTR To Reference External Data" section).

1.8.2 Connection Model

TCP/IP sockets use a connection model, where a client (*socket client*) seeks to establish a connection with a server (*socket server*). Once established, the socket client and server communicate over this connection in a peer-to-peer fashion by sending and receiving data streams over the established connection.

Thus, a 4GL socket client can:

- Establish a connection with a server.
- Write and read data on a connection.
- Disconnect from the server.

A 4GL socket server can:

- Notify Progress to listen and accept connections on a specified port.
- Have Progress notify the server of new connections from clients.
- Write and read data on a connection.
- Disconnect from the client.

1.8.3 Server Socket and Socket Objects

To enable 4GL applications to access sockets, Progress supports two types of objects:

- **Server socket objects** — A 4GL object that a socket server creates only to listen for client connection requests. A 4GL socket server receives notification of client connections in the form of events.
- **Socket objects** — A 4GL object that represents a TCP/IP socket. TCP/IP sockets are the communication endpoints of a connection. Both socket clients and servers use socket objects to read and write data on a connection. In the 4GL, a socket application can detect the arrival of data on a socket in the form of events. (However, this is not required.)

Using Sockets

Using methods on the socket object, a 4GL application can write and read any available data on a socket at any time, as long as the socket connection is active. Thus, the application can detect the arrival of data using events or not, depending on application requirements.

Both socket and server socket handles provide a variety of additional methods and attributes that you can use to monitor and control socket communications. For example, you can temporarily disable events on a socket object to run more efficiently when a client or server does not need to receive data on the connection. You can also temporarily or permanently disable events on a server socket when you no longer want the server to respond to client connection requests. Other methods and attributes allow you to set and monitor additional socket communications options and conditions.

1.8.4 4GL Socket Event Model

The 4GL socket event model includes two types of 4GL events:

- **CONNECT** — Posted on a server socket object when a client seeks to establish a socket connection. As part of generating this event, Progress creates a socket object for the server to communicate with the client on the new connection.
- **READ-RESPONSE** — Posted on a socket object when data is available to be read on the socket.

Using Events To Connect and Communicate

Like all 4GL events, these CONNECT and READ-RESPONSE events are handled in the context of an I/O blocking or PROCESS EVENTS statement. Figure 1–8 shows how two 4GL applications, one acting as a socket server and the other as a socket client, can use these events to communicate across a single connection.

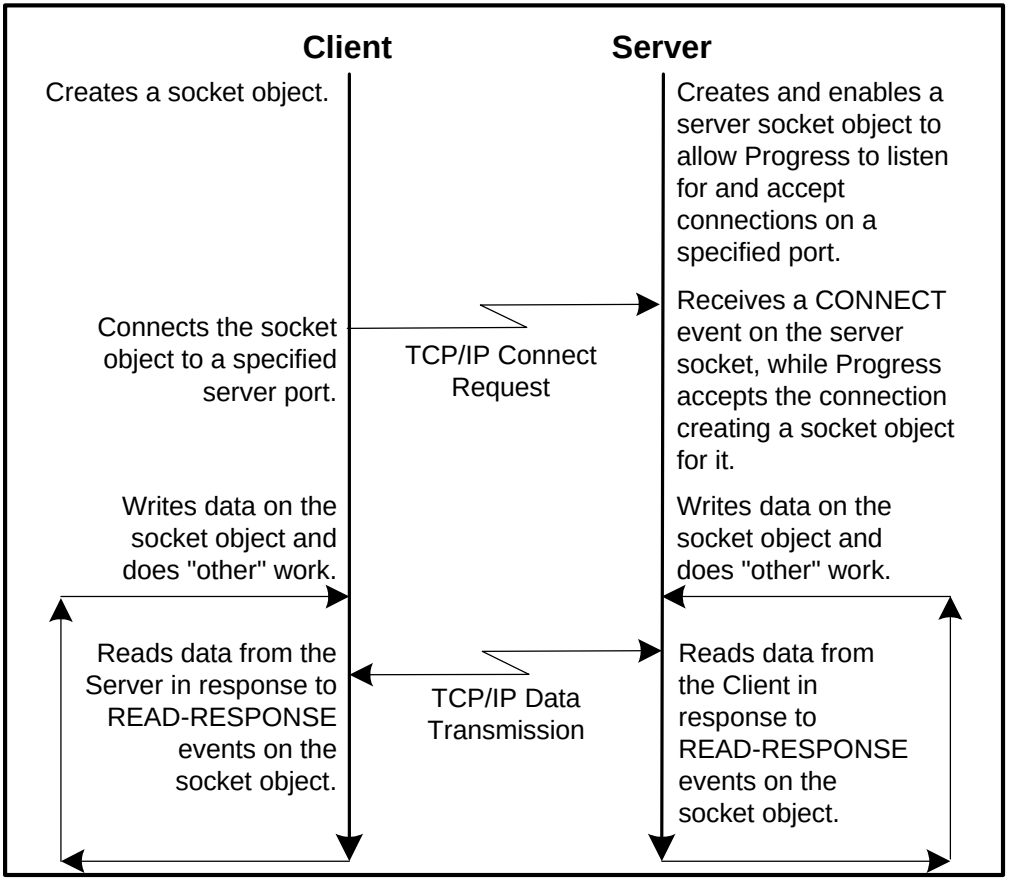


Figure 1–8: 4GL Socket Event Model

The server creates and enables a server socket object for listening on a specified port. The client creates a socket object and attempts to connect that socket to the server port used by the server socket. The server accepts the connection request and runs a specified CONNECT event procedure in response to the CONNECT event. This procedure receives a handle to a socket object that is implicitly created on the server for the connection. Once a connection is established, both the client and server can read and write data to each other using their connected socket objects.

The socket object that the client creates and the socket object created on the server in response to the CONNECT event both reference the same TCP/IP connection. Using the 4GL event model, the client and server can each receive notifications of data from the other within a READ-RESPONSE event procedure that runs in response to a READ-RESPONSE event on the socket object. The client can specify this event procedure any time after it creates its socket object. The server can specify this event procedure any time after it receives the socket object in the CONNECT event procedure.

1.8.5 Programming Requirements

To use sockets in the 4GL:

- Socket clients and servers must run locally or remotely in a TCP/IP network environment.
- A potential server must have a TCP/IP port available on which to listen for connections.
- A socket server can have only one server socket object enabled to listen for connections.
- AppServer and WebSpeed procedures cannot function as socket servers. They can only function as socket clients.
- A potential client must know the:
  - Hostname or IP address of the machine where the server it wants to connect to is running
  - TCP/IP port where the server is listening for connections

For more information on programming with sockets, see Chapter 10, “Sockets.”

1.9 XML

XML is a data format used for exchanging structured data over networks. It is hardware and software independent. XML documents are composed of a *prologue* and a *body*. The prologue contains general information about the document such as the XML version, the character encoding and the rules defining the document elements. The body of an XML document contains the data and the *markup* which encodes a description of the document’s logical structure.

1.9.1 XML and Progress

The Document Object Model (DOM) is an application programming interface (API) for XML documents. It defines the logical structure of documents and the way a document is accessed and manipulated. In the DOM specification, the term *document* is used in the broad sense to include many different kinds of information that might be stored in diverse systems. When you read an XML document via the DOM API, the DOM parser reads and parses the complete input document before making it available to the application.

Progress has defined an initial set of extensions to the Progress 4GL to allow the use of XML through the DOM interface. These extensions provide 4GL applications with the *basic* input, output, and low-level data manipulation capabilities required to use data contained in XML documents. They are not intended to provide access to the entire DOM interface, nor are they intended to include all the high-level constructs.

Note On DOM Compatibility With the 4GL

The DOM API is designed to be compatible with a wide range of programming languages, but the naming convention chosen by the World Wide Web Consortium (W3C) does not match what already exists in the Progress 4GL. In some cases, PSC elected to use the familiar names already used in the 4GL rather than the names given in the DOM specification. Similarly, where there are existing 4GL features that provide the same capability as the DOM interfaces, PSC has chosen to use the 4GL implementation rather than introduce new language features that match the DOM more closely.

Accessing XML With the Progress 4GL

The DOM presents documents as a hierarchy or tree of *node* objects that also implement other, more specialized interfaces. Progress implements the node interface as a Progress object. The document interface also inherits from the node interface. Progress has extended it to provide special methods.

This gives us two new object types in the 4GL for XML document manipulation:

- **X-DOCUMENT** — Which represents an entire XML document tree.
- **X-NODEREF** — Which represents a reference to a single node in the XML tree of the document.

NOTE: Progress also supports the Simple API for XML (SAX) interface to XML. For more information, see Chapter 12, “Simple API For XML (SAX),” in this book.

1.9.2 Requirements For Using XML

The minimum requirement with working with XML in Progress is a familiarity with using and manipulating Progress objects other than widgets and a knowledge of network communication via either the Web or sockets. Knowledge of HTML or SGML would also be very helpful.

1.10 The Progress SonicMQ Adapter

Application programmers can access Java Message Service (JMS) messaging from the Progress 4GL with the Progress SonicMQ Adapter. You can write 4GL applications that access the SonicMQ Adapter through a 4GL–JMS API that works the same on all platforms and in all configurations (GUI, character, AppServer, WebSpeed, and batch).

For information on requirements, see the "Requirements" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL," in this book.

2

Host Language Call Interface

This chapter describes the Progress Host Language Call (HLC) Interface in the following sections:

2.1 Using HLC

The Progress HLC Interface provides a way to call C language functions from within Progress 4GL applications. You can use Progress HLC to add portable custom features to Progress. For example, you can link in C functions that do the following:

- Add support for trigonometric and other special numeric functions to Progress
- Allow Progress to access and control multimedia devices, such as video equipment
- Allow Progress to directly access third-party proprietary data files, such as spreadsheet files

If you intend to port your extensions to other environments, use HLC. If you intend to build extensions for use with Windows only, use dynamic link libraries (DLL). If you intend to build extensions for use on UNIX platforms, use UNIX shared objects.

You can use shared libraries or DLLs with your Progress Windows applications to call routines from a 4GL procedure. An application links to these routines at run time rather than at build time, and shares the code with other applications. Any enhancement to a shared library or DLL immediately becomes available to your application without rebuilding.

For more information on using UNIX shared libraries or Windows DLLs with Progress, see Chapter 5, "Shared Library and DLL Support." See your Windows SDK documentation for details on building DLLs.

The HLC library functions provide an interface between C functions and Progress. From your C function, you can call HLC library functions that perform the following tasks:

- Read data from and write data to Progress 4GL shared or global variables
- Read data from and write data to Progress 4GL shared buffers
- Display Progress-like messages
- Control interrupts
- Perform timer-service operations

For more information on using HLC library functions, see the "Using HLC Library Functions" section later in this chapter. For details on each HLC library function, see Appendix A, "HLC Library Function Reference."

2.2 Overview Of HLC

You can use HLC to add virtually any feature to Progress that is written in C and that follows HLC programming rules. To use HLC, you should be familiar with the following topics:

- Using the Progress 4GL client.
- Using the PROBUILD End User Configuration (EUC) utility. See the *Progress Client Deployment Guide* for more information on PROBUILD.
- Designing, compiling, and linking C programs. See the C documentation for more information.

Figure 2–1 illustrates the steps to build an HLC executable.

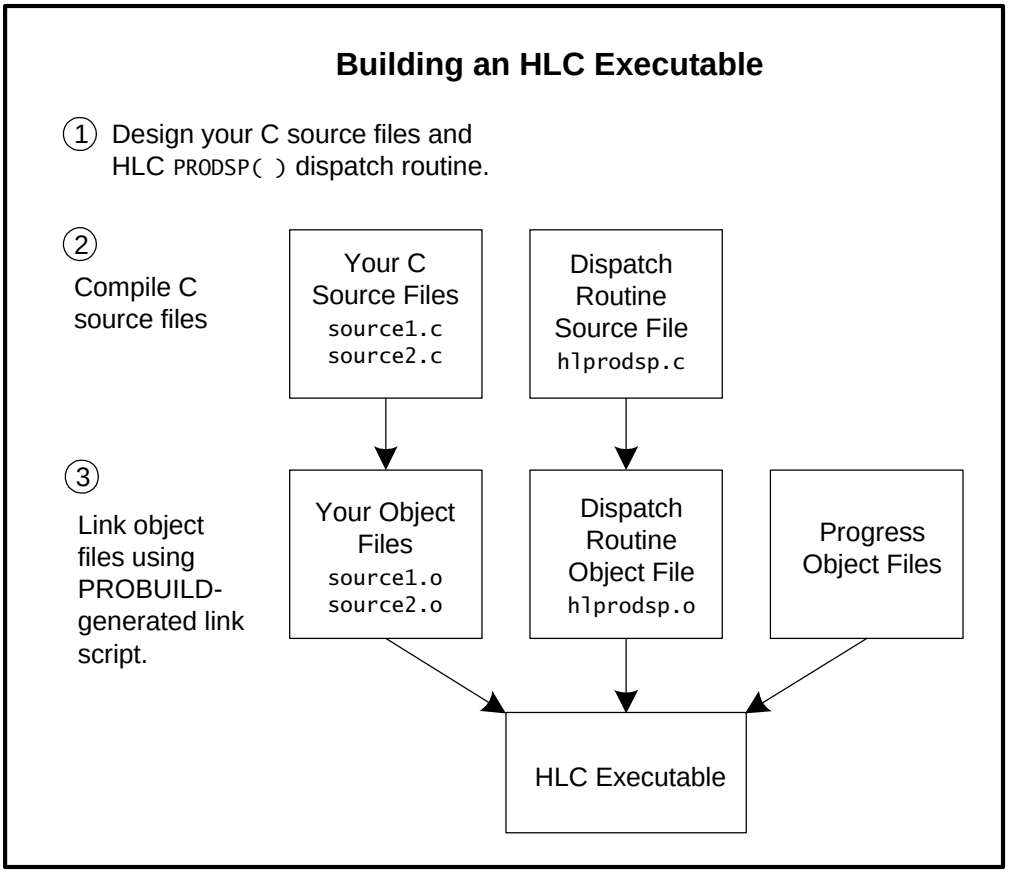


Figure 2–1: Steps To Build an HLC Executable

Follow these basic steps to build an HLC executable:

1 ♦ Write your C functions and update the `PRODSP()` dispatch routine contained in `hlprodsp.c`.

2 ♦ Compile your C routine source files and the `hlprodsp.c` dispatch routine source file.

3 ♦ Use the PROBUILD EUC utility to generate a link script for your executable.

4 ♦ Link your C object files, the `hlprodsp.c` dispatch routine object file, and Progress object files, using the link script that PROBUILD generates.

See the *Progress Client Deployment Guide* for information on the Progress PROBUILD EUC utility.

To use HLC you must build an HLC executable. An HLC executable is a Progress module with your C functions linked. Once you build an HLC executable, you can use the CALL statement to execute a linked C function from a Progress 4GL procedure.

2.2.1 Using the CALL Statement

Use the Progress 4GL CALL statement to execute a C function from a Progress procedure.

The CALL statement has the following syntax:

SYNTAX

```
CALL routine-identifier [ argument ... ]
```

The *routine-identifier* is the name the `PRODSP()` dispatch routine maps to an actual C function. The routine identifier is case sensitive; it must have the same letter case as its definition in the dispatch routine.

The *argument* is one or more arguments that you want to pass to the C function. Arguments passed to the C function must be read only. If you supply multiple arguments, separate them with spaces. If you separate them with other delimiters, such as commas, Progress passes the delimiters as arguments. Progress converts all arguments to C character strings before passing them to a C function; Progress passes them as an array of character pointers. Therefore, your C functions must expect null-terminated character strings and perform data conversions as necessary.

Figure 2–2 demonstrates how to run the CALL statement.

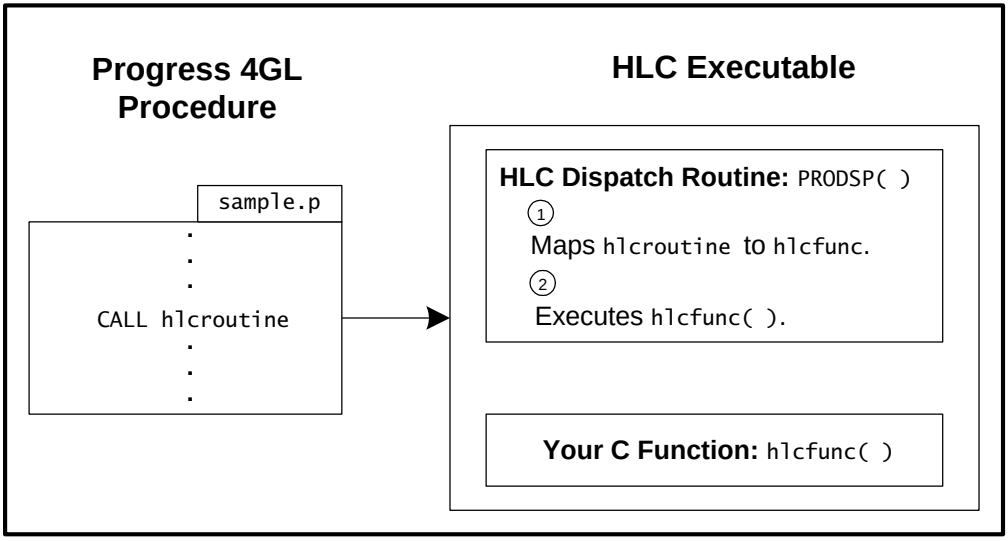


Figure 2–2: Running a CALL Statement

The following CALL statement executes `hlcroutine`, the routine identifier for your C function:

```
CALL hlcroutine.
```

When you use a CALL statement to invoke a routine that updates a shared buffer, you must make sure that a transaction is active at the time of the call. For more information, see the "Using a Library Function That Writes To a Shared Buffer" section.

The CALL statement transfers control to the HLC dispatch routine, `PRODSP()`, passing it your routine identifier and any arguments. The example in Figure 2–2 passes the hlcroutine routine identifier with no arguments to `PRODSP()`, which is located in `hlprodsp.c`.

You must modify the prototype `hlprodsp.c` file supplied with Progress HLC to define your routine identifiers and C function names. Based on definitions you have set up in `hlprodsp.c`, the `PRODSP()` dispatch routine maps the routine identifier to your C function, and calls it. The example in Figure 2–2 maps hlcroutine to the function `hlcfunc()`.

If the `PRODSP()` dispatch routine does not define the routine identifier or defines it with a different letter case, an error results.

2.2.2 Mapping Routine Identifiers Using PRODSP()

You map routine identifiers to C functions in the PRODSP() dispatch routine. Progress provides a prototype PRODSP() in the C source file, hlprodsp.c.

NOTE: The $DLC/probuild/hlc directory on UNIX and the %DLC%\probuild\hlc directory in Windows contain a prototype hlprodsp.c file. Do not modify this file. To make changes, copy it to a working directory and modify the copy.

Because Progress calls the PRODSP() dispatch routine, it must have the following declaration:

```
PRODSP(pfunnam, argc, argv)

     char   *pfunnam,     /* Name of function to call */
     int    *argc,        /* CALL statement argument count */
     char   *argv[],       /* CALL statement argument list */
```

The hlprodsp.c file shows routine-identifier hlcroutine being mapped to the C function hlcfunc() in PRODSP():

hlprodsp.c

```
#define FUNCTEST(nam, rout)  \
        if (strcmp(nam, pfunnam) == 0) \
            return rout(argc,argv);

/* PROGRAM: PRODSP
 *
 *   This is the interface to all C routines that
 *   Progress has associated 'CALL' statements to.
 */

long
PRODSP(pfunnam, argc, argv)

     char   *pfunnam;    /* Name of function to call */
     int     argc;       /* CALL statement argument count */
     char   *argv[];     /* CALL statement argument list */

{

    FUNCTEST("HLCROUTINE", hlcfunc);

    return 1;   /* Non-zero return code causes Progress error  */
}               /* condition if CALLed routine not found.    */
```

For each routine you add, you must include a call to the FUNCTEST macro. For example, to map two routine names, such as HLCROUTINE1 and HLCROUTINE2, to two corresponding C functions, such as hlcfunc1() and hlcfunc2(), you must include the following lines in PRODSP():

```
FUNCTEST("HLCROUTINE1", hlcfunc1);
FUNCTEST("HLCROUTINE2", hlcfunc2);
```

NOTE: The CALL statement and FUNCTEST declaration must use the same letter case for the routine identifier.

The FUNCTEST macro in hlprodsp.c has the following syntax:

SYNTAX

```
FUNCTEST ( "routine-identifier" , function-name ) ;
```

The *routine-identifier* is the name referenced by a CALL statement that identifies your C function. Enter the routine identifier as a character string surrounded by quotes. Since FUNCTEST does not convert case for the routine identifier, the case is significant.

The *function-name* is the name of the C function that the *routine-identifier* references.

The routine identifier and function name can have the same name:

```
FUNCTEST("hlcfunc", hlcfunc);
```

When you compile hlprodsp.c, the C compiler translates the FUNCTEST macro references to C code. For example:

```
FUNCTEST("HLCROUTINE", hlcfunc);
```

translates to:

```
if (strcmp("HLCROUTINE", pfunnam) == 0)
    return hlcfunc(argc,argv);
```

Therefore, when Progress invokes PRODSP() with the argument HLCROUTINE, it runs hlcfunc.

2.3 HLC Files and Directories

The default HLC installation process creates the directories listed in Table 2–1.

Table 2–1: HLC Directories

UNIX	Windows
`$DLC/probuild/hlc`	`%DLC%\probuild\hlc`
`$DLC/probuild/hlc/examples`	`%DLC%\probuild\hlc\examples`

These directories contain the files listed in Table 2–2.

Table 2–2: HLC Filenames

Filename	Description
`c`	Script to compile your C functions with the correct parameters to use with HLC.
`hlc.h`	HLC function library header file.
`hlprodsp.c`	Prototype HLC dispatch routine file.

The examples directory contains HLC example files for an oil tank demo application called Tank, described later in this chapter.

2.4 Writing C Functions

Figure 2–3 shows the relationship between the PRODSP() dispatch routine that Progress supplies and your C functions.

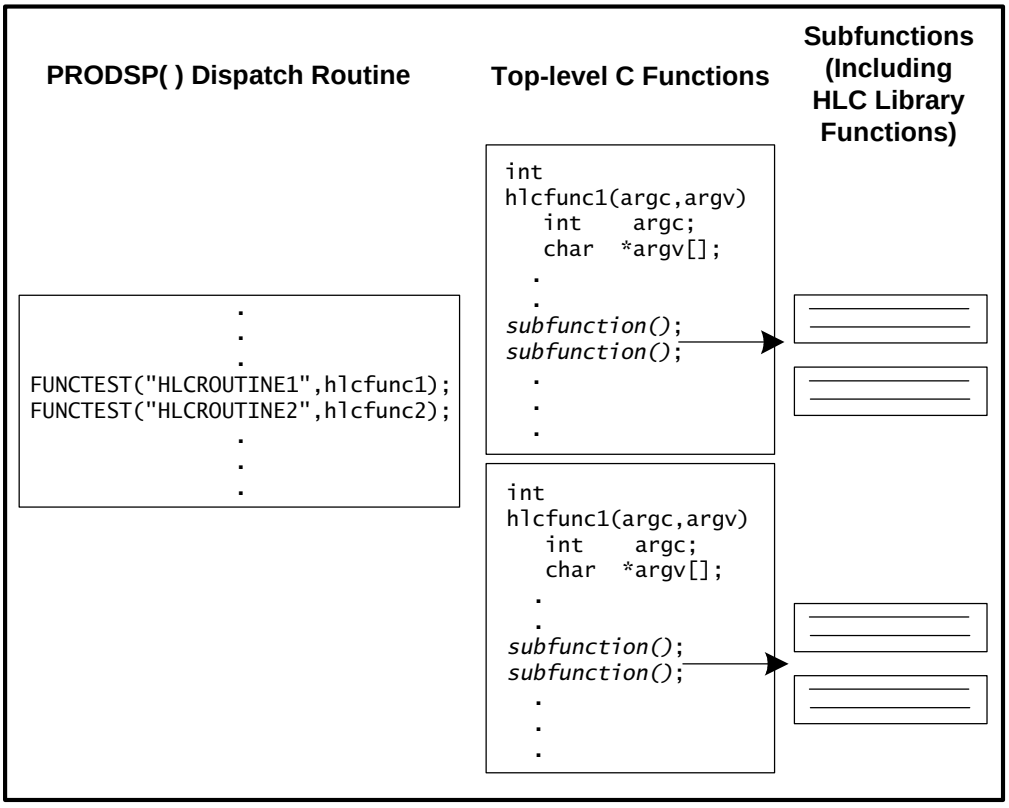


Figure 2–3: Relationship Between PRODSP() and Your C Functions

In general, your C functions can perform the following tasks:

- Call HLC library functions to read data from and write data to Progress shared buffers and variables. HLC library functions are C functions that Progress provides to access the Progress application environment from a user-written C function.
- Perform operations using data from both Progress and other sources. Other sources might include optical scanners, process monitor and control devices, spreadsheets, and other devices or applications not supported directly by Progress.

Your C functions consist of functions called directly from PRODSP() (top-level functions), and functions called directly or indirectly from the top-level functions (subfunctions). All top-level functions must return control to Progress, regardless of the subfunctions called.

2.4.1 Top-level C Function Declaration

If you use the FUNCTEST macro provided in the prototype hlprodsp.c file, your top-level function must use the following interface:

SYNTAX

```
long function-name ( arcg, argv )
  int   argc;
  char *argv[];
```

The length and content of the function name must follow the conventions that your C compiler and linker require. The *argc* and *argv* parameters follow the rules defined for *argc* and *argv* passed by the standard main() C function. If your top-level function must use a different interface, either write alternative dispatch code for it or rewrite FUNCTEST to meet your particular needs.

For more information on the FUNCTEST macro, see the "Mapping Routine Identifiers Using PRODSP()" section.

2.4.2 Returning Error Codes From a Top-level Function

There are two ways to return errors from a top-level function:

- Return from your top-level function with a non-zero value. Progress receives the non-zero value from PRODSP() and raises an error condition.
- Before returning from your top-level function, set a Progress shared variable to a specific value using an HLC library function.

If you use the first technique, the PRODSP() dispatch routine returns a long data type value to Progress. Progress uses this value as a return code to determine whether the HLC routine dispatched by PRODSP() was successful. If the return code is 0, Progress considers the routine successful and continues processing. If the return code is non-zero, Progress considers the routine unsuccessful and raises an error condition. See the *Progress Programming Handbook* for more information on error processing.

The FUNCTEST macro ensures that the value returned from your top-level function is passed on as the return code of PRODSP(). However, you must make sure that your top-level function returns an appropriate value for the return code. For example, when your top-level function runs successfully, make sure that the last statement executes the equivalent of the following:

```
return 0;
```

If you use the second method, you can set unique error codes that you can test from Progress and respond to each type of error as needed.

The first technique provides an efficient way to return to Progress when your HLC function encounters an unexpected error, such as an unsuccessful HLC library call. The second technique provides more fine-grained error processing, and allows your Progress procedure to handle conditions specific to your application.

2.4.3 Naming C Functions

You must name all external functions (entry points) so that your function names do not conflict with Progress entry point names.

On many systems, if you have an entry point in your code with the same name as a Progress entry point, the linker detects duplicate entry point symbols and displays an error message. However, on some systems the linker replaces one entry point with the other, and does not display a message.

No Progress entry point ends with _USR. Therefore, use this syntax to guarantee that your function names do not conflict with Progress entry point names:

SYNTAX

```
function-name_USR ( ... )
```

The *function-name* is any name compatible with your development environment. For example, you might have a function called calc_USR().

2.4.4 C Function Portability

C code is portable from one operating system to another when you can compile, link, and run it without change on either operating system. If you plan to port your HLC application, make your C functions as portable as possible. The following tips can help increase the portability of your C code:

- Avoid operating-system-specific calls. Proposed UNIX-style portable operating system standards exist (POSIX, XOPEN), but their acceptance is limited. Progress runs on operating systems that do not support these standards.
- If you must make operating-system-specific calls, hide them behind a standard application interface of your own design. If possible, confine the interface definition and all operating-system-specific code to a single source module, where you can modify it easily.
- Use the UNIX lint utility if it is available on your system. The lint utility flags C language statements in your code that are potential sources of portability problems. See your system documentation for information on using lint or its equivalent.
- When you pass a value to a function, cast it to the data type the function expects. For example the following function call casts the number 23 as a long:

```
sample_function((long)23);
```

- Do not use C functions when you can use standard Progress 4GL code.
- Do not write your own function if there is an HLC library function that provides the same functionality.

For more information on developing portable Progress applications, see the *Progress Portability Guide*.

2.5 Avoiding Common HLC Errors

To ensure compatibility between your HLC functions and Progress, consider the following when writing your functions:

- An HLC library function does not always return character and decimal data as null-terminated strings. For example, the *pvar* parameter of the `prordc()` function contains character data without null termination. To terminate such a string with the null character, you must add it yourself.
- When you check the value of a field or variable that HLC returns, check for the unknown value (?). The field might not always contain a valid value.
- If you return from your top-level function without setting an appropriate return code value, Progress might raise an error condition unexpectedly. See the "Returning Error Codes From a Top-level Function" section for more information.
- Use the `promsgd()` HLC library function to display messages directly to the terminal. Using the standard C function `printf()` might produce unexpected results when used to send raw data to a terminal.

2.6 Memory Allocation

Use `malloc()` and `free()` to allocate and deallocate memory within a C function. Before returning to Progress, deallocate all memory that you allocate in the function.

CAUTION: If you allocate too much memory in an HLC function and leave it allocated after returning to Progress, your Progress application might not have enough memory for its own needs. If Progress runs out of memory at any point in the 4GL application, it aborts and returns to the operating system with an error.

2.7 Data Size

In order to run multiple instances of the Progress executable in Windows, the static data size must not exceed 64K. The Progress executable is a large model application that meets this requirement. However, if your HLC objects add excessive static data to the module, a linker error such as the following occurs:

```
DGROUP Greater than 64K.
```

Follow these basic steps to avoid this problem:

1. Dynamically allocate any large buffers or arrays.
2. Move your static text strings and constants into a resource file or into the code segment using _based pointers, as shown in the following example:

```
char _based(segname("_CODE")) mytext[] = "string of text";
```

See the Windows C Compiler documentation for details.

2.8 Using HLC Library Functions

The HLC library functions provide an interface between your C functions and Progress. From your C functions, you can call HLC library functions that perform the following tasks:

- Read data from and write data to Progress 4GL shared or global variables
- Read data from and write data to Progress 4GL shared buffers
- Display Progress-like messages
- Control interrupts
- Perform timer-service operations

2.8.1 Accessing Progress Data

This section describes how to use HLC library functions to access Progress data. For example, you can use the `prordbi()` function to read an integer field in a shared buffer. Use the following guidelines when accessing Progress data with HLC library functions.

CAUTION: If you do not follow these guidelines, you can permanently damage your Progress database.

- **Use correct data types for the parameters you pass to HLC library functions.** In general, C compilers do not verify whether the data types of the parameters you provide in a function call agree with the parameter data type definitions specified in the function declaration.

- **Use correct values for the parameters you pass to HLC library functions.** For example, HLC functions that access Progress shared buffers use the *fhandle* parameter. The *fhandle* parameter has the integer data type. You must set *fhandle* correctly to read from or write to the correct location in the buffer.

- **Use pointer variables properly.** Many HLC library functions that access Progress use pointer variable parameters. Improper use of pointer variables can cause you to overwrite random locations in memory, with potentially hazardous results.

The following example demonstrates these guidelines by defining shared buffer custbuf within a 4GL procedure:

```
DEFINE NEW SHARED BUFFER custbuf FOR customer.
FIND first custbuf.
CALL subfunc1.
```

Later in your 4GL code, execute a CALL statement that calls a C function. Within the C function, you read the cust-num field. The cust-num field is an integer field in the customer table, for which shared buffer hlcbuff is defined, as follows:

```
#include "hlc.h"
subfunc1()
{
  int    ret, index, unknown;
  char    message[80];
  long    cnum;
  int    fhandle;

index = 0;
unknown = 0;

fhandle = profldix ("custbuf", "cust-num");
ret = prordbi ("custbuf", fhandle, index, &cnum, &unknown);
if (ret||unknown)
{
  sprintf (message, "prordbi fatal ret = %d unknown %d", ret, unknown);
  promsgd (message);
}
sprintf(message, "customer.cust-num was %ld", cnum);
promsgd(message);
return 0;
}
```

The `prordbi()` function reads an integer field contained in a shared buffer. To determine the HLC library function to use, see the "Function Summary" section in Appendix A, "HLC Library Function Reference."

The `prordbi()` function has the following syntax:

SYNTAX

```
int prordbi ( pbufnam , fhandle , index , pvar , punknown )
  char *pbufname ;
  int   fhandle ;
  int   index ;
  long *pvar ;
  int  *punknown ;
```

The *pbufnam* parameter points to the name of the specified shared buffer. You supply the name from your Progress application.

The *fhandle* input parameter is the field handle that `profldix()` returns for the specified field.

The *index* input parameter specifies an index value for an array field. If the field is a scalar, you must set the value of *index* to 0.

The *pvar* output parameter points to a long where `prordbi()` returns the value of the specified integer field.

The *punknown* output parameter points to an integer where `prordbi()` returns 1 if the field has the unknown value, and returns 0 otherwise.

Figure 2–4 shows a call being made to `prordbi()` that illustrates HLC programming guidelines:

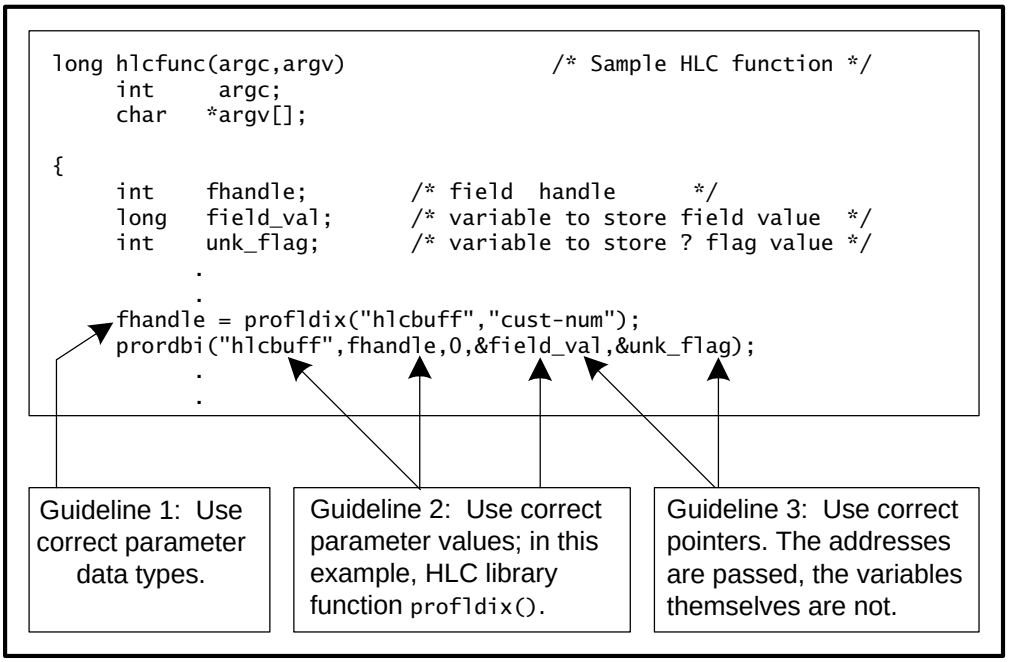


Figure 2–4: Example HLC Library Function Call

2.8.2 Data Type Conversion

HLC data type conversion happens in two directions:

- From C language data types to Progress data types. This occurs when you write to a Progress shared variable or shared buffer.
- From Progress data types to C language data types. This occurs when you read from a Progress shared variable or shared buffer.

Converting C Language Data Types To Progress Data Types

When you use an HLC library function to write data to a Progress shared buffer or shared variable, the HLC library function converts the C language data type passed to the function to the appropriate Progress data type.

For example, the `prowtbd()` HLC library function allows you to write to a date field in a shared buffer. Among the parameters you pass to `prowtbd()` are *year*, *month*, and *day*, all defined as integer in your C code. The `prowtbd()` function converts these separate C language integer values into a single Progress date value.

Converting Progress Data Types To C Language Data Types

When you use an HLC library function to read data from a Progress shared buffer or shared variable, the HLC library function converts the Progress data type of the buffer or variable to a C language data type.

For example, the `prordbd()` HLC library function allows you to read from a Progress date field in a shared buffer. Among the parameters you pass to `prordbd()` are *pyear*, *pmonth*, and *pday*, all defined as pointers to integer in your C code. The `prordbd()` function converts the single date value within the buffer to separate C language integer values and inserts the values in the memory locations to which *pyear*, *pmonth*, and *pday* point.

See Appendix A, "HLC Library Function Reference," for details on the parameters each HLC library function uses and the data types of these parameters.

2.8.3 Calling an HLC Library Function

Any C source file that calls an HLC library function must include the `hlc.h` header file at the beginning of the file, as shown in the following example for UNIX:

```
#include"$DLC/probuild/hlc/hlc.h"
```

NOTE: This statement assumes you installed HLC in the default `$DLC/probuild/hlc` directory for UNIX.

2.8.4 Timer Services

Timer services provide functions that allow your application to wait for or respond to the completion of a specified time interval on the system clock.

Any HLC function that requires timer services should not use the `sleep()` or other operating system functions to access the system clock. Access the following HLC library functions for timer services on UNIX: `prosleep()`, `proevt()`, `prowait()`, and `procncel()`. Access `prosleep` for timer services in Windows. The `proevt()`, `prowait()`, and `procncel()` timer services are available only on UNIX. See Appendix A, "HLC Library Function Reference," for information on these functions.

The timer-service functions provide all the tools you need to determine whether a timer is expired. The UNIX-specific program fragment shows examples of timer-service functions `proevt()`, `prockint()`, and `procncel()`, as shown in Figure 2–5.

```
char timerflag;
int seconds = 60;        /* choose the time interval you desire */

timerflag = 0;
proevt(seconds, &timerflag) /* set a timer */

while(  (ret=<systemcall>) == -1 && errno == EINTR  )
{   /* a signal interrupted the system call  */
    if (prockint())  { /* it was CTRL-C or equiv. */ ... }
    if (timerflag)   { /* the timer event occurred */ }
}
procncel(&timerflag);  /* in case the timer did not expire */
if (ret < 0)  { /* the system call failed */ }
```

Figure 2–5: Timer-service Functions On UNIX

2.8.5 User Interrupt Handling

The HLC function library contains an interrupt-handling function, `prockint()`, that allows you to check when a user presses the **STOP** key (**CONTROL-C** on UNIX; **CONTROL-BREAK** in Windows).

Use `prockint()` in place of your own interrupt-handling function. Using your own interrupt-handling function during an HLC call could interfere with how Progress handles interrupts once you return from the call.

2.8.6 Using a Library Function That Writes To a Shared Buffer

When you use a CALL statement to invoke a routine that writes to a shared buffer, you must make sure that a transaction is active at the time of the call. For example, place the CALL statement in a DO TRANSACTION block, as shown in the following code fragment:

```
DEFINE NEW SHARED VARIABLE errcode AS INTEGER.
DEFINE NEW SHARED BUFFER newcust FOR CUSTOMER.
FIND FIRST newcust.

/* 1 */
DO TRANSACTION:
   .
   .
   .
/* 2 */
    CALL HLCROUTINE1.
    IF errcode = 2 THEN DO:
        MESSAGE "Unable to read newcust".
        UNDO, RETRY.
    END.
   .
   .
   .
/* 3 */
END.
```

These notes explain the transaction block:

1. The transaction begins.
2. The CALL statement is defined.
3. The transaction ends.

The explicit DO TRANSACTION statement is not the only way to create a transaction in Progress. For example, you create an implicit transaction when you use the UPDATE statement within a FOR EACH block. See the *Progress Programming Handbook* for more information on transactions.

If your C function calls an HLC library function that writes to a shared buffer, (for example, `prowtbc()`), and no transaction is active, the library function returns a non-zero return code value that indicates an error has occurred and a Progress error condition results. Use an ON ERROR phrase within your Progress 4GL code to handle Progress error conditions.

2.8.7 Passing Error Codes Back To Progress

You can use HLC library functions to pass a specific error code from your C function to Progress and test it for a specific value. To do this, define a shared variable within your Progress procedure to hold the error code value. Within your C function, set the shared variable with an HLC library function before returning to Progress. In the code fragment above, the shared variable `errcode` holds the error code value.

2.9 Building an HLC Executable

After designing the CALL statements, supporting C functions, and the `PRODSP()` dispatch routine for your application, follow these steps to build an HLC executable:

1 ♦ On UNIX operating systems, set up your environment using the Progress BUILDENV utility. This command sets the search paths and options required with your compiler and linker to build Progress executables.

2 ♦ Compile your copy of the `hlprodsp.c` source file that contains the HLC dispatch routine and the source files that contain your C functions. For more information, see the "Compiling C Source Files" section.

3 ♦ Use the PROBUILD utility to generate a link script for your executable. The PROBUILD utility allows you to do the following:

- Select the Progress product and configurable elements (including HLC) that you want to build into your executable.
- Enter the filenames of your object files and `hlprodsp.o` in the dialog box that PROBUILD displays. The PROBUILD utility inserts the filenames in the link script it generates.

4 ♦ Link your executable with the link script that PROBUILD generates. This step produces your HLC executable.

For more information on building Progress executables, see the *Progress Client Deployment Guide*.

2.10 HLC Applications On UNIX Systems

This section explains how to handle special features of UNIX operating systems for HLC applications, including:

- Handling raw disk I/O
- Handling terminal I/O
- Handling abnormal exits

2.10.1 Handling Raw Disk I/O

On UNIX systems that do not provide a synchronous write instruction, Progress uses raw disk I/O when writing to some of its key files. An HLC link script automatically modifies the permissions and ownership of the executable it creates. However, you might have to supply a root password when running the link script. If your Progress executable does not have proper permissions, UNIX displays a message similar to the following when you attempt to execute the new module:

```
Unable to use raw disk I/O
```

2.10.2 Handling Terminal I/O

Character-mode systems support two basic modes of terminal I/O:

- **Raw** — Terminal I/O without any operating system processing. Set your terminal to raw mode using the [-]raw option of the `stty` command.
- **Cooked** — Terminal I/O that the operating system processes. Set your terminal to cooked mode using either the [-]raw or [-]cooked option of the `stty` command.

In raw mode, the system reads input characters immediately and passes them to an application without any interpretation, and without sending them to the display as they are entered. Also, the system does no preprocessing or postprocessing of output characters.

In cooked mode, the system interprets input characters according to the following terminal input functions that the UNIX system defines: ERASE, KILL, INTR, QUIT, SWITCH, and EOT. Also, UNIX systems provide terminal-specific postprocessing, such as defining special characters and character mapping, echoing input to the display, etc.

Using the proscopn(), prosccls(), and promsgd() Functions

In character mode, Progress typically uses raw terminal I/O, but supplies HLC library functions for switching between raw and cooked mode. Ideally, leave the terminal set to raw mode and use the `promsgd()` function to display information to the user. This provides a uniform appearance for messages.

If necessary, you can use cooked mode for handling the screen display in character mode. In this case, use the `proscopn()` function to enable cooked mode. Before returning to Progress, re-enable raw mode with a call to `prosccls()`.

In character mode, the `promsgd()` function displays up to two messages before displaying a "Press space bar to continue" status message. In raw mode, the user can press **SPACEBAR** as prompted. In cooked mode, the user must press **RETURN** after `promsgd()` displays the status message. To use `promsgd()` in cooked mode, use the Progress PAUSE statement to change either the status message or the way the user interacts with the display.

For example, to change the status message to tell the user to press **RETURN**, place the following PAUSE statement anywhere before the first HLC call that invokes `promsgd()`:

```
PAUSE BEFORE-HIDE MESSAGE "Press [RETURN] to continue"
                      .
                      .
                      .
CALL HLC-MESSAGE-ROUTINE.
```

To cause the display to change two seconds after displaying a message without user input, use the following statement:

```
PAUSE 2 BEFORE-HIDE NO-MESSAGE.
                      .
                      .
                      .
CALL HLC-MESSAGE-ROUTINE.
```

In this case, you must specify the interval to pause. See the PAUSE Statement reference entry in the *Progress Language Reference* for more information.

Run the HLC demo application to see examples that include a number of `promsgd()` calls illustrating these techniques.

In graphical interfaces, `promsgd()` displays messages in an alert box. Raw and cooked terminal I/O, and the `proscopn()`, `proclear()`, and `prosccls()` functions apply only to Progress running in character interfaces.

2.10.3 Handling Abnormal Exits

In character mode, if you abort to the operating system during application testing, or otherwise must abort from your production version, make sure your HLC application does not terminate with raw terminal I/O. Set the terminal to cooked terminal I/O before exiting. This prepares the terminal for UNIX operation, as shown in the following example:

```
/*
 * Example of a fatal error exit for an HLC application.
 *
 * Note how the terminal is returned to cooked mode before
 * exiting. Note also that exiting in this fashion can be
 * hazardous to your database. (This function should only
 * occur in program development phase for convenience in the
 * debugging process.)
 */

void hl_fexit (code)
{
  proscopn(); /* restore terminal to cooked mode */
  exit (1);
}
```

CAUTION: An abnormal exit function might be useful during testing, but you should never include it in a production executable. Since it bypasses Progress shutdown processing, this function can cause irrecoverable database damage.

Progress automatically restores cooked-mode terminal I/O before exiting. If your application aborts in raw mode, the user can enter commands but might not see the result.

Follow these steps to reset a character-mode terminal in raw mode at a UNIX shell prompt:

1 ♦ Press **CONTROL-J** several times. Shell prompts appear on the display.

2 ♦ If the `stty` command on your version of UNIX has a sane option, enter the following command, followed by **CONTROL-J**:

```
stty sane
```

If your `stty` command does not have a sane option, enter the following command, followed by **CONTROL-J**:

```
stty echo icanon icrnl opost
```

2.11 Compiling C Source Files

Compile the `hlprodsp.c` dispatch routine source file and your C routine source files. For example, the following command compiles source files `source.c` and `hlprodsp.c`:

```
c source.c hlprodsp.c
```

The `c` script is in `$DLC/probuild/hlc` on UNIX and `%DLC%\probuild\hlc` in Windows. It contains a number of options you might want to modify for your particular application environment.

2.12 Example HLC Application

This section explains how to use the Progress HLC tank demo application. This application is a prepackaged application that you can use to verify that your HLC environment is set up correctly. The C code and sample output for this example are provided in the `$DLC/probuild/hlc/examples` directory on UNIX, and in the `%DLC%\probuild\hlc\examples` directory in Windows.

Set up your environment and compile this example before you try to run it to verify that your compiler and the PROBUILD utility are working correctly. This also helps you learn how to use HLC in a controlled environment.

This sample application uses Progress to keep track of oil storage tanks. Your Progress procedure calls a C program, `AVCALC`, to calculate the available capacity for a given tank. Figure 2–6 shows that the tanks are cylindrical, with their axes parallel to the level ground.

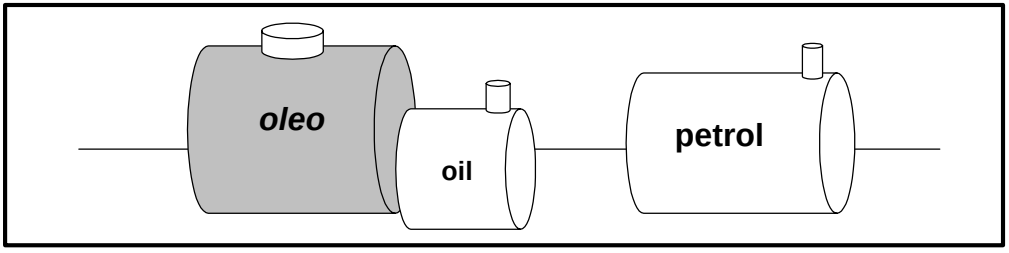


Figure 2–6: Tank Positioning and Orientation

To calculate the available capacity (empty portion) of the tank, you need to know the tank's diameter, length, and current level of oil. Use the variables in the following formula to calculate the available tank volume:

$$length \cdot r \cdot \left[\frac{p}{2} + \sqrt{1 - \left(1 - \frac{level}{r}\right)^2} \left(1 - \frac{level}{r}\right) + \sin^{-1}\left(1 - \frac{level}{r}\right) \right]$$

Where:

r	Radius of the tank
length	Length of the tank
level	Level of oil in the tank

For this example, assume there is a tank table for this application that contains the following decimal fields:

radius	Radius of tank
tlength	Length of tank
depth	Level of oil—must be between 0 and (2 · radius)
tavail	Available volume in tank

In addition to these fields, the tank-id character field is used as the primary index.

Figure 2–7 shows the Data Dictionary report for the tank table.

```
           Default field order: yes
09/10/93              Progress Data Dictionary Report               Page   1
Database: tank
                                    tank File
                                    =========
                   (Flat file containing oil tank information)

Delete Validation
    Criterion:
      Message:
    Field        Type  Ext Dec Format                          Init
    ------------ ----- --- --- ------------------------------ ----------
*   tank-id      char          x(8)
    radius       dec         2 ->>,>>9.99                     0
    tlength      dec         2 ->>,>>9.99                     0
    depth        dec         2 ->>,>>9.99                     0
    tavail       dec         2 ->>,>>9.99

Index Name   Unique Field Name   Seq Ascending abbreviate
  ------------ ------ ------------ --- --------- ----------
# tank         yes    tank-id        1 yes       yes

Field Validation Criteria, Validation Messages
------------------------------------------------------------------------------
depth        :  depth le 2 * (input radius)
                Depth cannot exceed diameter of tank

Help Messages
------------------------------------------------------------------------------
tank-id      :  Tank identification number
radius       :  Radius of oil tank
tlength      :  Length of oil Tank
depth        :  Depth of oil in tank.
tavail       :  Available Volume In Tank


                        Data Dictionary Report Legend
             * - Indicates that a field participates in an index
             # - Indicates the primary index for a database file
             M - Indicates that a field is mandatory
```

Figure 2–7: Progress Data Dictionary Report For Tank Application

A C function calculates the tavail field from the other three decimal fields (radius, tlength, depth). Figure 2–8 shows the Progress procedure that invokes the C function by calling the HLC routine, AVCALC.

```
/* calculate available volume for each tank */
DEFINE NEW SHARED BUFFER tankbuf FOR tank.
FOR EACH tankbuf:
  DISPLAY
    radius SPACE(3) tlength SPACE(3) depth SPACE(3)
    tavail WITH CENTERED TITLE "Tank Table".
END.
PAUSE 0.
VIEW FRAME tank-before.
HIDE ALL.

FOR EACH tankbuf:
  DO TRANSACTION:
    CALL AVCALC.
  END.
  DISPLAY
    radius SPACE(3) tlength SPACE(3) depth SPACE(3)
    tavail WITH CENTERED TITLE "After Calculation".
END.
```

Figure 2–8: Progress Procedure Calling HLC Routine AVCALC

Make a copy of the HLC dispatch routine, `hlprodsp.c`, and name it `tankdsp.c`. Modify the routine so that an entry appears for AVCALC, which calls the C subroutine `hlvcalc`.

Example hlprodsp.c shows the modifications to the `tankdsp.c` routine:

hlprodsp.c

```
#define FUNCTEST(nam, rout)  \
        if (strcmp(nam, pfunnam) == 0) \
            return rout(argc,argv);
/* PROGRAM: PRODSP
 *
 *   This is the interface to all C routines that
 *   Progress has associated 'call' statements to.
 */
long
PRODSP(pfunnam, argc, argv)
    char   *pfunnam;
    /* Name of function to call */
    int     argc;
    char   *argv[];{
    /* Interface to 'tank' example  */
    FUNCTEST ("AVCALC", hlvcalc);

    return 1;
}
```

The following procedure shows the code for the demo program, `hlvcalc.c`. The program extracts the radius, length, and level fields from the shared buffer tank, calculates the available volume, and updates the tavail field in the shared buffer tank with the number calculated:

hlvcalc.c *(1 of 3)*

```
 *    Obtains the height, radius and oil level for the tank
 *    from the shared buffer "tankbuf".
 *
 *    Calculates the remaining available volume
 *
 *    Update the avail field in the shared buffer "tankbuf" with the
 *    number calculated.
 */
#define  BUFLEN   100
#include <math.h>
#include "hlc.h"
/*NOTE: M_PI_2 is pi/2 constant which may be defined in math.h */
#ifndef M_PI_2
#define M_PI_2   1.570796327
#endif

extern double asin();

char *fieldnm[] = { "tlength", "depth", "radius"};
char message[80];
int
hlvcalc()
{
   char     buffer[BUFLEN];
   int      unknown = 0, index = 0, varlen = BUFLEN, actlen;
   int      ret;
   double   length, depth, radius, avail;
   int      i;
   int      fldpos;
   double   temp1, temp2;  /* used to simplify calculation */
   /* first, obtain the length, depth and radius from */
   /* the shared buffer "tankbuf".                    */
   for (i = 0; i < 3; ++i)
   {
      fldpos = profldix("tankbuf", fieldnm[i]);
      if (fldpos < 0)
      {
        sprintf(message, "profldix failed on %s for field %s",
                "tankbuf", fieldnm[i]);
        promsgd(message);
        return 1;
       }
```

hlvcalc.c *(2 of 3)*

```
      ret = prordbn("tankbuf", fldpos, index,
                    buffer, &unknown, varlen, &actlen);
      if (ret)
      {
         sprintf(message, "prordbn failed accessing %s . %s",
                  "tankbuf", fieldnm[i]);
         promsgd(message);
         return 1;
      }
      /* if one of the fields is unknown, set avail field */
      /* to the unknown value                             */
      if (unknown)
      {
         fldpos = profldix("tankbuf", "tavail");
         if (fldpos < 0)
         {
            sprintf(message, "profldix failed on %s for field %s",
                     "tankbuf", "tavail");
            promsgd(message);
            return 1;
         }
         ret = prowtbn("tankbuf", fldpos, index, buffer, unknown);
         if (ret)
         {
            sprintf(message, "prowtbn failed, ret = %d", ret);
            promsgd(message);
            return 1;
         }
         return 0;
      }

      /* convert the character string obtained from */
      /* Progress into a decimal number             */
      buffer[actlen] = '\0';

      switch (i)
      {
         case 0:
                  length = atof(buffer); break;
         case 1:
                  depth  = atof(buffer); break;
         case 2:
                  radius = atof(buffer); break;

         default:
                  break;
      }
```

hlvcalc.c *(3 of 3)*

```
/* Now, calculate the available volume                  */
/* NOTE: M_PI_2 is pi/2 constant defined in math.h */
#ifndef M_PI_2
#define M_PI_2 1.57
#endif
temp1 = 1.0 - depth/radius;
temp2 = temp1 * sqrt(1.0 - temp1 * temp1) + asin(temp1);
avail = length * radius * radius * (temp2 + M_PI_2);

/* Now, put this value in the tavail field in the */
/* "tankbuf" shared buffer                        */
/* get the double into character format */
sprintf(buffer, "%.2f", avail);
fldpos = profldix("tankbuf", "tavail");
if (fldpos < 0)
{
       sprintf(message, "profldix failed on %s for field %s",
                                      "tankbuf", fieldnm[i]);
       promsgd(message);
       return 1;
}
ret = prowtbn("tankbuf", fldpos, index, buffer, unknown);
if (ret)
       {
           sprintf(message, "prowtbn failed, ret = %d", ret);
           promsgd(message);
           return 1;
       }
return 0;
```

2.12.1 Running the Sample Application On Windows

To verify that your HLC application is installed correctly, run the sample application by completing the following steps:

1 ♦ Create a working directory, then go to it.

NOTE: Before you develop your own applications, move `c.bat` to a directory that is in your path and move `hlc.h` to your include file directory or current working directory.

2 ♦ Copy the example HLC files to your working directory, as follows:

```
copy %DLC%\probuild\hlc\examples\*.*
```

3 ♦ Create an empty database:

```
prodb tankdb empty.
```

4 ♦ Load and run the Progress programs `loaddb.p` and `initdb.p`.

5 ♦ Compile the C source files to create object files in your current directory:

```
c tankdsp.c hlvcalc.c
```

6 ♦ Run the Progress PROBUILD EUC utility using `tankdsp` and `hlvcalc` as object files for the HLC application.

See the *Progress Client Deployment Guide* for more information on the PROBUILD utility.

7 ♦ Copy the `.def` and `.res` files to your current working directory:

```
copy %DLC%\probuild\_prowin.*
```

8 ♦ Link your test HLC module using the link script PROBUILD generates:

```
link @_prowin.lnk
```

9 ♦ Link in the resource files:

```
rc -k _prowin.res
```

10 ♦ Run `hltank.p`.

After you run `hltank.p`, verify that the results you obtain are correct by comparing them to the following results that list the input values for radius, tlength, and depth, and the correct output for tavail:

```
              After Calculation

   radius      tlength        depth       tavail
 ----------   ----------   ----------   ----------
    5.00        20.00         0.00       1,570.80
    5.00        20.00         5.00         785.40
    5.00        20.00        10.00           0.0
   10.00        20.00        10.00       3,141.59
   10.00        20.00        13.00       1,959.84
```

2.12.2 Running the Sample Application On UNIX

To verify that your HLC application is installed correctly, run the sample application by completing the following steps:

1 ♦ Run the `buildenv` script provided in the `$DLC/probuild/eucapp` directory to set up your HLC environment and to set environment variables for the C compiler and linker.

2 ♦ Create a working directory, then go to it.

3 ♦ Copy the example HLC files to your working directory, as follows:

```
cp $DLC/probuild/hlc/examples/* .
```

NOTE: Before you develop your own applications, move `c` and `buildenv` to a directory that is in your path and move `hlc.h` to your include file directory or your current working directory.

4 ♦ Enter **testhlc** on the command line.

The `testhlc` script creates an empty database, compiles the C source files to create object files in your current directory, links your test HLC module, and runs a small sample application.

After you run `hltank.p`, verify that the results you obtain are correct by comparing them with the following results, which list the input values for radius, tlength, and depth, and the correct output for tavail:

```
              After Calculation

   radius      tlength        depth       tavail
 ----------   ----------   ----------   ----------
     5.00        20.00         0.00      1,570.80
     5.00        20.00         5.00        785.40
     5.00        20.00        10.00          0.0
    10.00        20.00        10.00      3,141.59
    10.00        20.00        13.00      1,959.84
```

Figure 2–9 shows the `testhlc` script in UNIX.

```
  #!/bin/sh
  #  testhlc -- single-user test of HLC

①
  DLC=${DLC-/usr/dlc};export DLC
  DLCDB=${DLCDB-$DLC};export DLCDB

②
  SOURCEDB=$DLCDB/empty
  DBNAME=hlcdemo
  rm -fr $DBNAME.*
  prodb $DBNAME $SOURCEDB

③
  echo Now compiling *.c ...
  c *.c

④
  echo Now linking test hlc module ...
  ./ldhlcrx
⑤
  ./_progres $DBNAME -1 -p hldemo.p

  if [ -r hlcout ]
  then
      if cmp -s hlcout savehlc
      then
           echo "** HLC Installation Test has Completed Successfully."
  else
           diff hlcout savehlc >testhlc.diff
           echo "** HLC Installation Test reports conflicts."
           echo "   Please look at files:"
           echo "      'hlcout.dif', 'hlcout' & 'savehlc'"
  fi
```

Figure 2–9: UNIX testhlc Script

These notes explain the blocks in Figure 2–9:

1. Sets up your UNIX environment.

2. Creates and starts an empty database.

3. Compiles all the `.c` files necessary to run the sample application.

4. Links and loads the object files.

   NOTE: This link script is specific to the sample application. For your own application development, use the PROBUILD utility to automatically generate a tailored link script.

5. Starts a Progress session and runs the `hldemo.p` procedure.

2.12.3 Source Code Listings

This section shows some of the source files provided in the `$DLC/probuild/hlc/examples` directory.

This procedure automatically loads the data definitions for the sample application:

loaddb.p

```
hide all.
display  skip(2)
        "  *********   LOADING the database ********* " skip(1)
        "  *********    please stand by....."
        with title "WELCOME TO HLC DEMO PROGRAM"
                 row 5 centered frame hdr.
output to hldemout.
run prodict/load_df.p ("customer.df").
run prodict/load_df.p ("agedar.df").
run prodict/load_df.p ("monthly.df").
run prodict/load_df.p ("tank.df").
/* create signal file to indicate success */
if opsys = "unix" then do:
   unix  cat >hlcsigfile & .
end.
hide all.
```

This is the Progress prototype file in which you define the application's HLC routine identifiers and the corresponding C function names:

hlprodsp.c

```
#define FUNCTEST(nam, rout)  \
        if (strcmp(nam, pfunnam) == 0) \
            return rout(argc,argv);

/* PROGRAM: PRODSP
 *
 *   This is the interface to all C routines that
 *   Progress has associated 'call' statements to.
 */

long
PRODSP(pfunnam, argc, argv)

     char   *pfunnam;    /* Name of function to call */
     int     argc;
     char   *argv[];

{
/* Interface to installation test */
    FUNCTEST( "SUBVRD", subvrd)
    FUNCTEST( "SUBVWT", subvwt)
    FUNCTEST( "SUBFRD", subfrd)
    FUNCTEST( "SUBFWT", subfwt)
    FUNCTEST( "SUBFIX", subfix)
    FUNCTEST( "SUBVIX", subvix)
    FUNCTEST( "SUBARG", subarg)
    FUNCTEST( "SUBCLR", subclr)
    FUNCTEST( "SUBCLS", subcls)
    FUNCTEST( "SUBINT", subint)
/* Interface to 'screen' examples */
    FUNCTEST ("EXAMPLE1", example1);
    FUNCTEST ("EXAMPLE2", example2);
/* Interface to 'tank' example  */
    FUNCTEST ("AVCALC", hlvcalc);

    return 1;
}
```

The following example is the front end to the sample test application:

hldemo.p *(1 of 2)*

```
run loaddb.p.

input from terminal.
output to terminal.
define var answer as char format "9".
define var choice as char init "1,2,3,4".
define var last1 as log init yes.
define var last2 as log init yes.

form   skip(4)
       space (4)
         "  1  -   Installation Test" skip(1) space(4)
         "  2  -   TANK Capacity Calculation demo  "  skip(1) space(4)
         "  3  -   Screen raw/cooked demo " skip (1)   space(4)
         "  4  -   Exit Session" skip(2) space(4)
         " Enter example selection ===> "  answer no-label
              with centered title " EXAMPLE MENU " frame example.

form last1 last2 with frame lastcheck.
repeat.
  hide all.
  update answer auto-return
  validate(lookup(answer,choice) <> 0,"Enter one of choices displayed.")
     with frame example.
  hide all.
if answer = "1" then
   do:
/*  Check to see if need to reinitialize database. */
    if last1 then
    do:
       last1 = yes.
       last2 = no.
       run initdb.p.
       hide all.
    end.
    last1 = yes.
    run hltest.p.
   end.
```

hldemo.p *(2 of 2)*

```
   else if answer = "2" then
     do:
/*  Check to see if need to reinitialize database. */
      if last2 then
      do:
         last2 = yes.
         last1 = no.
         run initdb.p.
         hide all.
      end.
      last2 = yes.
      run hltank.p.
     end.
    else if answer = "3" then
          run hlscreen.p.
    else if answer = "4" then
          quit.

end.
hide all.
quit.
```

3

System Clipboard

The *system clipboard* is a feature provided on most window systems that allows the user to transfer data between one widget (or application) and another using standard mouse and keyboard operations. Each application typically provides some form of program support for how these operations interact with it. You can provide this support in a Progress application using the CLIPBOARD system handle.

Progress supports clipboard operations between Progress and other applications in Windows. In character mode, Progress supports clipboard operations within a Progress application. For more information on how clipboard operations work in Windows, see the *Microsoft Windows User's Guide*.

This chapter contains the following sections:

3.1 CLIPBOARD System Handle

The CLIPBOARD system handle allows you to transfer data between the window system clipboard and your Progress application. Using the CLIPBOARD attributes, you can paste (read) data from the system clipboard to a Progress field or variable, and copy or cut (write) data from a field or variable to the clipboard. These cut, copy, and paste data transfers are the basic *clipboard operations* typically provided by the system clipboard to the user.

In Progress, you also have a choice of two data transfer modes to implement these operations — single-item transfers and multiple-item transfers. In *single-item transfers*, a single write to the clipboard immediately replaces all data in the clipboard, and a single read from the clipboard returns all data in the clipboard to the Progress application. In *multiple-item transfers*, you can format the data transfer into multiple rows of multiple items. Each write to the clipboard adds an item to a tab- and newline-separated list of clipboard items; each read from the clipboard returns one tab- or newline-separated item to your Progress application. This mode is especially useful to allow users to transfer aggregate units of data, in one step, between Progress and other applications (such as spreadsheets) that also support aggregate clipboard operations in a similar way.

These data transfers are accomplished with the help of the CLIPBOARD handle attributes listed in Table 3–1.

Table 3–1: CLIPBOARD Handle Attributes

Attribute	Type	Readable	Setable
AVAILABLE-FORMATS	CHARACTER	√	–
ITEMS-PER-ROW	INTEGER	√	√
MULTIPLE	LOGICAL	√	√
NUM-FORMATS	INTEGER	√	–
TYPE	CHARACTER	√	–
VALUE	CHARACTER	√	√

The following sections provide an overview of the CLIPBOARD handle attributes. For a complete description of the CLIPBOARD handle and its attributes, see the CLIPBOARD Handle reference entry in the *Progress Language Reference*.

3.1.1 AVAILABLE-FORMATS Attribute

The AVAILABLE attribute returns a comma-separated list of the available formats for the data stored in the clipboard. The supported formats include:

- **PRO_TEXT** — Specifies the standard text format on your system (CF_TEXT for Windows).
- **PRO_MULTIPLE** — Specifies that the data in the clipboard contains tab or newline characters, and thus can be read as multiple items.

3.1.2 ITEMS-PER-ROW Attribute

The ITEMS-PER-ROW attribute specifies how many tab-separated items are formatted in a newline-separated row for multiple-item writes to the clipboard. This attribute has no effect on multiple-item reads from the clipboard. The user or program that originally moves the data to the clipboard must format the data according to how your application expects to read it.

3.1.3 MULTIPLE Attribute

Setting the MULTIPLE attribute to TRUE starts a multiple-item transfer. Setting it to FALSE ends the multiple-item transfer and readies the clipboard for single-item transfers. After multiple writes, a FALSE setting transfers the formatted list of items to the clipboard and resets the ITEMS-PER-ROW attribute to 1. After multiple reads, a FALSE setting allows you to restart reading from the first item after resetting the attribute to TRUE.

3.1.4 NUM-FORMATS Attribute

The NUM-FORMATS attribute returns the number of data formats available for reading data from the clipboard. If there is no data in the clipboard, the value is 0.

3.1.5 TYPE Attribute

The TYPE attribute returns the widget type of the CLIPBOARD handle, which is the standard type for system handles, "PSEUDO-WIDGET". This attribute has no effect on data transfers, and is essentially used for documentation.

3.1.6 VALUE Attribute

The VALUE attribute provides access to the system clipboard data. Set the attribute to the value of a field or variable you want to cut or copy to the clipboard. To cut, set the source data item to the unknown value (?) or the null string ("") after you set the attribute. Assign the value of the attribute to a field or variable to which you want to paste data from the clipboard. If there is no data in the clipboard or you read the last item in a multiple read, the attribute returns the unknown value to the data item.

A single-item write to this attribute immediately replaces all previous data in the clipboard. A multiple-item write to this attribute appends the data item to a buffered list of items. Once you set the MULTIPLE attribute to FALSE, the CLIPBOARD handle formats the list according to the value of ITEMS-PER-ROW and replaces all previous clipboard data with it. (Note that in Windows, the clipboard can store a maximum of 64K of data). Both single-item and multiple-item reads are nondestructive to data in the clipboard.

3.2 Single-item Data Transfers

Each single-item data transfer moves data between the Progress data item and the clipboard. During a paste operation, all data stored in the clipboard is transferred to the data item. During a cut or copy operation, the value of the Progress data item replaces any and all data in the clipboard.

Clipboard operations are typically invoked by Cut, Copy, and Paste options on an Edit menu. You can implement these operations in a general way using the FOCUS system handle (for more information, see the FOCUS Handle reference entry in the *Progress Language Reference*). This allows you to program two types of actions that:

- Determine what clipboard operations are available (enabled and disabled) at any point.
- Specify how each clipboard operation is implemented when it is available.

3.2.1 Enabling and Disabling Clipboard Operations

You can enable and disable clipboard operations based on the type of widget that currently has the input focus (FOCUS:TYPE attribute). For example, you might disable the pasting (inserting) of values into a selection list, but enable the copying of selected items from the list. You typically configure your clipboard operations in the trigger block of the MENU-DROP event for your Edit menu. This ensures that you enable or disable menu options based on the latest selection action that the user has committed in the current input widget (for example, selected a radio set or a range of text in an editor widget).

The following code example suggests a possible scenario for enabling and disabling cut, copy, and paste operations. It defines an Edit menu (EditMenu) with Cut, Copy, and Paste options assigned to the corresponding menu items EM_Cut, EM_Copy, and EM_Paste. When the user opens the Edit menu (ON MENU-DROP OF MENU EditMenu), the code determines the available options from the state of the field-level widget that has the current input focus.

For example, if an editor widget has the input focus (FOCUS:TYPE = "EDITOR"), then the Cut and Copy options are available only if the user has text selected within the widget (LENGTH(FOCUS:SELECTION-TEXT) > 0). If a radio set, selection list, slider, or toggle box has the input focus, then only the Copy option is enabled. You could make the Cut and Paste options meaningful for radio sets and selection lists, for example, by recreating dynamic radio sets or removing and inserting items in selection lists.

Although this chapter provides useful examples, your code to determine the available clipboard operations can vary widely depending on your application.

```
DEFINE MENU EditMenu
    MENU-ITEM EM_Cut       LABEL "&Cut "
    MENU-ITEM EM_Copy      LABEL "C&opy "
    MENU-ITEM EM_Paste     LABEL "&Paste ".

ON MENU-DROP OF MENU EditMenu DO:
    IF FOCUS:TYPE = "EDITOR" THEN DO:
       MENU-ITEM EM_Cut:SENSITIVE IN MENU EditMenu =
          IF  LENGTH(FOCUS:SELECTION-TEXT) > 0
          THEN TRUE
          ELSE FALSE.
       MENU-ITEM Em_Copy:SENSITIVE IN MENU EditMenu =
          IF  LENGTH(FOCUS:SELECTION-TEXT) > 0
          THEN TRUE
          ELSE FALSE.
       MENU-ITEM EM_Paste:SENSITIVE IN MENU EditMenu =
          IF CLIPBOARD:NUM-FORMATS > 0
          THEN TRUE
          ELSE FALSE.
    END.
    ELSE IF FOCUS:TYPE = "RADIO-SET"      OR
            FOCUS:TYPE = "SELECTION-LIST" OR
            FOCUS:TYPE = "SLIDER"         OR
            FOCUS:TYPE = "TOGGLE-BOX" THEN DO:
       MENU-ITEM EM_Cut:SENSITIVE IN MENU EditMenu = FALSE.
       MENU-ITEM Em_Copy:SENSITIVE IN MENU EditMenu = TRUE.
       MENU-ITEM Em_Paste:SENSITIVE IN MENU EditMenu = FALSE.
    END.
    ELSE IF FOCUS:TYPE = "FILL-IN" THEN DO:
       MENU-ITEM EM_Cut:SENSITIVE IN MENU EditMenu =
          IF LENGTH(FOCUS:SCREEN-VALUE) > 0
          THEN TRUE
          ELSE FALSE.
       MENU-ITEM Em_Copy:SENSITIVE IN MENU EditMenu =
          IF LENGTH(FOCUS:SCREEN-VALUE) > 0
          THEN TRUE
          ELSE FALSE.
       MENU-ITEM EM_Paste:SENSITIVE IN MENU EDitMenu =
          IF CLIPBOARD:NUM-FORMATS > 0
          THEN TRUE
          ELSE FALSE.
    END.
    ELSE DO:
        MENU-ITEM EM_Cut:SENSITIVE IN MENU EditMenu = FALSE.
        MENU-ITEM EM_Copy:SENSITIVE IN MENU EditMenu = FALSE.
        MENU-ITEM EM_Paste:SENSITIVE IN MENU EditMenu = FALSE.
    END.
END. /* ON MENU-DROP IN EditMenu */
```

3.2.2 Implementing Single-item Transfers

You can implement each clipboard operation based on the type of widget that currently has the input focus (FOCUS:TYPE) and the state of its text selection and other attributes. For example, you can decide whether to copy all or part of an editor widget value by the values of the SELECTION-START and SELECTION-END attributes. You typically implement each clipboard operation in the trigger block of the CHOOSE event for the corresponding Edit menu option. In this way, the user can only perform a clipboard operation associated with an Edit menu option that is enabled.

The following code example implements the clipboard operations enabled by the code example in the previous section. Note that for editor widgets (FOCUS:TYPE = "EDITOR"), if the user has text selected, the procedure transfers data between the clipboard and the SELECTION-TEXT rather than the VALUE attribute itself.

Paste Operations

For example, the paste operation (ON CHOOSE OF MENU-ITEM EM_Paste) replaces only the selected text (rather than the whole text) in an editor widget with the data in the clipboard. (In a fill-in widget, paste operations always replace all data in the widget.)

Cut Operations

For cut operations (ON CHOOSE OF MENU-ITEM EM_Cut), the procedure sets the appropriate widget attribute (SELECTION-TEXT or VALUE) to the empty string after transferring the data to the clipboard. The corresponding text disappears from the display as the Cut operation completes.

Copy Operations

Copy operations (ON CHOOSE OF MENU-ITEM EM_Copy) are similar to cut operations except that they leave the FOCUS data unchanged. However, if the data to be copied is a radio set, the example assumes that the character value of the radio set label visible on the display (FOCUS:LABEL) is what the user wants to copy rather than its value (FOCUS:VALUE). This is a useful implementation where the radio set represents an integer and the FOCUS:VALUE attribute contains a right-justified integer string:

```
DEFINE VARIABLE Stat AS LOGICAL.
DEFINE MENU EditMenu
    MENU-ITEM EM_Cut       LABEL "&Cut "
    MENU-ITEM EM_Copy      LABEL "C&opy "
    MENU-ITEM EM_Paste     LABEL "&Paste ".

ON CHOOSE OF MENU-ITEM EM_Cut IN MENU EditMenu DO:
    IF FOCUS:TYPE = "EDITOR" THEN DO:
        IF FOCUS:SELECTION-START <> FOCUS:SELECTION-END THEN DO:
            CLIPBOARD:VALUE = FOCUS:SELECTION-TEXT.
            Stat = FOCUS:REPLACE-SELECTION-TEXT("").
        END.
        ELSE DO:
            CLIPBOARD:VALUE = FOCUS:SCREEN-VALUE.
            FOCUS:SCREEN-VALUE = "".
        END.
    END.
    ELSE DO: /* For FILL-IN */
        CLIPBOARD:VALUE = FOCUS:SCREEN-VALUE.
        FOCUS:SCREEN-VALUE = "".
    END.
END. /* ON CHOOSE OF MENU-ITEM EM_Cut */
ON CHOOSE OF MENU-ITEM EM_Copy IN MENU EditMenu DO:
    IF FOCUS:TYPE = "EDITOR" THEN
        IF FOCUS:SELECTION-START <> FOCUS:SELECTION-END THEN
            CLIPBOARD:VALUE = FOCUS:SELECTION-TEXT.
        ELSE
            CLIPBOARD:VALUE = FOCUS:SCREEN-VALUE.
    ELSE IF FOCUS:TYPE = "RADIO-SET" THEN
        CLIPBOARD:VALUE = ENTRY(LOOKUP(FOCUS:SCREEN-VALUE,
                                       FOCUS:RADIO-BUTTONS) - 1,
                                FOCUS:RADIO-BUTTONS).
    ELSE IF FOCUS:TYPE = "TOGGLE-BOX" THEN
        IF FOCUS:SCREEN-VALUE = "yes" THEN
            CLIPBOARD:VALUE = FOCUS:LABEL + " selected.".
        ELSE
            CLIPBOARD:VALUE = FOCUS:LABEL + " not selected.".
    ELSE /* For FILL-IN */
        CLIPBOARD:VALUE = FOCUS:SCREEN-VALUE.
END. /* ON CHOOSE OF MENU-ITEM EM_Copy */
ON CHOOSE OF MENU-ITEM EM_Paste IN MENU EditMenu DO:
    IF FOCUS:TYPE = "EDITOR" THEN DO:
        IF FOCUS:SELECTION-START <> FOCUS:SELECTION-END THEN
            Stat = FOCUS:REPLACE-SELECTION-TEXT(CLIPBOARD:VALUE).
        ELSE
            Stat = FOCUS:INSERT-STRING(CLIPBOARD:VALUE).
    END.
    ELSE /* For FILL-IN */
        FOCUS:SCREEN-VALUE = CLIPBOARD:VALUE.
END. /* ON CHOOSE OF MENU-ITEM EM_Paste */
```

3.2.3 Single-item Transfer Example

The following procedure uses the clipboard operation implementation described in the previous section. It both demonstrates the capabilities of that design and serves as a primer for other design alternatives. (To run `e-clpbrd.p` in character mode, comment out all statements that reference MainWindow and assign the MENU MainMenu:HANDLE attribute to the CURRENT-WINDOW:MENUBAR attribute.)

e-clpbrd.p *(1 of 5)*

```
DEFINE VARIABLE Stat AS LOGICAL.
DEFINE VARIABLE MainWindow AS WIDGET-HANDLE.
DEFINE VARIABLE Editor  AS CHARACTER
       VIEW-AS EDITOR SIZE 20 BY 4 SCROLLBAR-VERTICAL.
DEFINE VARIABLE Fillin AS CHARACTER FORMAT "x(20)".
DEFINE VARIABLE TogDemo AS LOGICAL EXTENT 2
       INITIAL ["FALSE", "TRUE"]
       LABEL "Pick-Me1",
             "Pick-Me2"
       VIEW-AS TOGGLE-BOX.
DEFINE VARIABLE Radios AS INTEGER INITIAL 3
       LABEL "Time-O-Day"
       VIEW-AS RADIO-SET RADIO-BUTTONS
       "1pm", 1,
       "2pm", 2,
       "3pm", 3.
DEFINE VARIABLE Slider1 AS INTEGER
       VIEW-AS SLIDER MAX-VALUE 100 MIN-VALUE 10 LABEL "Slide Me:".
DEFINE VARIABLE Slider2 AS INTEGER
       VIEW-AS SLIDER MAX-VALUE 1000 MIN-VALUE 100 VERTICAL LABEL
      "Slide Me:".
```

e-clpbrd.p *(2 of 5)*

```
DEFINE VARIABLE SelectList AS CHARACTER
       VIEW-AS SELECTION-LIST
       SINGLE SIZE 23 BY 7
       LIST-ITEMS
       "Line 1",
       "Line 2",
       "Line 3",
       "Line 4",
       "Line 5",
       "Line 6",
       "Line 7",
       "Line 8",
       "Line 9",
       "Line 10",
       "Line 11",
       "Line 12",
       "Line 13",
       "Line 14",
       "Line 15".DEFINE SUB-MENU FileMenu
    MENU-ITEM FM_New       LABEL "&New"
    MENU-ITEM FM_Open      LABEL "&Open... "
    RULE
    MENU-ITEM FM_Save      LABEL "&Save "
    MENU-ITEM FM_Save_as   LABEL "Save &As... "
    RULE
    MENU-ITEM FM_Exit      LABEL "E&xit ".

DEFINE SUB-MENU EditMenu
    MENU-ITEM EM_Cut       LABEL "&Cut "
    MENU-ITEM EM_Copy      LABEL "C&opy "
    MENU-ITEM EM_Paste     LABEL "&Paste ".

DEFINE MENU MainMenu
    MENUBAR
    SUB-MENU  FileMenu     LABEL "&File "
    SUB-MENU  EditMenu     LABEL "&Edit "

DEFINE BUTTON b_OK         LABEL "   OK   ".
DEFINE BUTTON b_Cancel     LABEL " CANCEL ".
```

e-clpbrd.p *(3 of 5)*

```
FORM
    "Enter Text Here" AT ROW 1 COLUMN 2
    Editor AT ROW 2 COLUMN 2
    "Fill In Here" AT ROW 1 COLUMN 39
    Fillin AT ROW 2 COLUMN 39
    TogDemo[1] AT ROW 2 COLUMN 25
    TogDemo[2] AT ROW 4 COLUMN 25
    Radios AT ROW 7 COLUMN 2
    "Selection List" AT ROW 6 COLUMN 22
    SelectList AT ROW 7 COLUMN 17
    Slider2 AT ROW 7 COLUMN 45
    Slider1 AT ROW 12 COLUMN 43
    b_OK AT ROW 12 COLUMN 2
    b_Cancel AT ROW 14 COLUMN 2
    SKIP (0.5)
WITH FRAME MainFrame NO-LABEL
     CENTERED
     WIDTH 62.

ON CHOOSE OF b_OK IN FRAME MainFrame MESSAGE "OK pressed".
ON CHOOSE OF b_Cancel IN FRAME MainFrame MESSAGE "CANCEL pressed".
ON CHOOSE OF MENU-ITEM FM_Exit IN MENU FileMenu STOP.

/****** Begin Clipboard Code ******/
ON MENU-DROP OF MENU EditMenu DO:
    IF FOCUS:TYPE = "EDITOR" THEN DO:
       MENU-ITEM EM_Cut:SENSITIVE IN MENU EditMenu =
          IF  LENGTH(FOCUS:SELECTION-TEXT) > 0
          THEN TRUE
          ELSE FALSE.
       MENU-ITEM Em_Copy:SENSITIVE IN MENU EditMenu =
          IF  LENGTH(FOCUS:SELECTION-TEXT) > 0
          THEN TRUE
          ELSE FALSE.
       MENU-ITEM EM_Paste:SENSITIVE IN MENU EditMenu =
          IF CLIPBOARD:NUM-FORMATS > 0
          THEN TRUE
          ELSE FALSE.
    END.
```

e-clpbrd.p *(4 of 5)*

```
    ELSE IF FOCUS:TYPE = "RADIO-SET"      OR
            FOCUS:TYPE = "SELECTION-LIST" OR
            FOCUS:TYPE = "SLIDER"         OR
            FOCUS:TYPE = "TOGGLE-BOX" THEN DO:
       MENU-ITEM EM_Cut:SENSITIVE IN MENU EditMenu = FALSE.
       MENU-ITEM Em_Copy:SENSITIVE IN MENU EditMenu = TRUE.
       MENU-ITEM Em_Paste:SENSITIVE IN MENU EditMenu = FALSE.
    END.
    ELSE IF FOCUS:TYPE = "FILL-IN" THEN DO:
       MENU-ITEM EM_Cut:SENSITIVE IN MENU EditMenu =
          IF LENGTH(FOCUS:SCREEN-VALUE) > 0
          THEN TRUE
          ELSE FALSE.
       MENU-ITEM Em_Copy:SENSITIVE IN MENU EditMenu =
          IF LENGTH(FOCUS:SCREEN-VALUE) > 0
          THEN TRUE
          ELSE FALSE.
       MENU-ITEM EM_Paste:SENSITIVE IN MENU EDitMenu =
          IF CLIPBOARD:NUM-FORMATS > 0
          THEN TRUE
          ELSE FALSE.
    END.
    ELSE DO:
        MENU-ITEM EM_Cut:SENSITIVE IN MENU EditMenu = FALSE.
        MENU-ITEM EM_Copy:SENSITIVE IN MENU EditMenu = FALSE.
        MENU-ITEM EM_Paste:SENSITIVE IN MENU EditMenu = FALSE.
    END.
END. /* ON MENU-DROP IN EditMenu */

ON CHOOSE OF MENU-ITEM EM_Cut IN MENU EditMenu DO:
    IF FOCUS:TYPE = "EDITOR" THEN DO:
        IF FOCUS:SELECTION-START <> FOCUS:SELECTION-END THEN DO:
            CLIPBOARD:VALUE = FOCUS:SELECTION-TEXT.
            Stat = FOCUS:REPLACE-SELECTION-TEXT("").
        END.
        ELSE DO:
            CLIPBOARD:VALUE = FOCUS:SCREEN-VALUE.
            FOCUS:SCREEN-VALUE = "".
        END.
    END.
    ELSE DO: /* For FILL-IN */
        CLIPBOARD:VALUE = FOCUS:SCREEN-VALUE.
        FOCUS:SCREEN-VALUE = "".
    END.
END. /* ON CHOOSE OF MENU-ITEM EM_Cut */
```

e-clpbrd.p *(5 of 5)*

```
ON CHOOSE OF MENU-ITEM EM_Copy IN MENU EditMenu DO:
    IF FOCUS:TYPE = "EDITOR" THEN
        IF FOCUS:SELECTION-START <> FOCUS:SELECTION-END THEN
            CLIPBOARD:VALUE = FOCUS:SELECTION-TEXT.
        ELSE
            CLIPBOARD:VALUE = FOCUS:SCREEN-VALUE.
    ELSE IF FOCUS:TYPE = "RADIO-SET" THEN
        CLIPBOARD:VALUE = ENTRY(LOOKUP(FOCUS:SCREEN-VALUE,
                                       FOCUS:RADIO-BUTTONS) - 1,
                                FOCUS:RADIO-BUTTONS).
    ELSE IF FOCUS:TYPE = "TOGGLE-BOX" THEN
        IF FOCUS:SCREEN-VALUE = "yes" THEN
            CLIPBOARD:VALUE = FOCUS:LABEL + " selected.".
        ELSE
            CLIPBOARD:VALUE = FOCUS:LABEL + " not selected.".
    ELSE /* For FILL-IN */
        CLIPBOARD:VALUE = FOCUS:SCREEN-VALUE.
END. /* ONCREATE ON CHOOSE OF MENU-ITEM EM_Paste IN MENU EditMenu DO:
    IF FOCUS:TYPE = "EDITOR" THEN DO:
        IF FOCUS:SELECTION-START <> FOCUS:SELECTION-END THEN
            Stat = FOCUS:REPLACE-SELECTION-TEXT(CLIPBOARD:VALUE).
        ELSE
            Stat = FOCUS:INSERT-STRING(CLIPBOARD:VALUE).
    END.
    ELSE /* For FILL-IN */
        FOCUS:SCREEN-VALUE = CLIPBOARD:VALUE.
END. /* ON CHOOSE OF MENU-ITEM EM_Paste */
/****** End Clipboard Code ******/

WINDOW MainWindow
    ASSIGN
        X = 0 Y = 0
        MENUBAR = MENU MainMenu:HANDLE
        TITLE = "CLIPBOARD SUPPORT".

CURRENT-WINDOW = MainWindow.

ON WINDOW-CLOSE OF MainWindow STOP.

ENABLE ALL WITH FRAME MainFrame.
STATUS DEFAULT "Widgets and the Clipboard".

WAIT-FOR CHOOSE OF b_Cancel IN FRAME MainFrame.
DELETE WIDGET MainWindow. CHOOSE OF MENU-ITEM EM_Copy */
```

3.3 Multiple-item Data Transfers

Each multiple-item data transfer moves data between one or more Progress data items and the clipboard. During a paste operation, all data stored in the clipboard is transferred to the data items. During a cut or copy operation, the values of the selected Progress data items replace any and all data in the clipboard.

Multiple-item clipboard operations are typically invoked by Cut, Copy, and Paste options on an Edit menu, just like single-item operations. There are two basic techniques you can use to implement multiple transfers in a procedure:

- **Widget-based transfers** — Provide a selection mode that allows the user to select eligible widgets for the selected clipboard operation. After the user selects and confirms the widgets that are valid for the operation, the operation proceeds.
- **Data-based transfers** — Provide options to transfer data directly between the Progress database and the clipboard. This is the most common type of multiple-item data transfer.

These are the essential tasks to implement any multiple-item data transfer.

1. Determine the clipboard operation to perform and the data items to participate in the operation.
2. Set the CLIPBOARD handle MULTIPLE attribute to TRUE. For Cut/Copy (write) operations, set the ITEMS-PER-ROW attribute to the number of items in each line of data written to the clipboard.
3. For each data item, assign the appropriate data item value (screen or record buffer) to the VALUE attribute for Cut/Copy operations or assign the VALUE attribute to the data item for Paste (read) operations.
4. Set the MULTIPLE attribute to FALSE to complete the operation. This resets the item pointer to the beginning of the clipboard for a Cut/Copy operation and writes the item-formatted data to the clipboard for a Paste operation. (This also resets the ITEMS-PER-ROW attribute.)

3.3.1 Widget-based Transfers

The techniques for implementing widget-based multiple-item transfers are very similar to those used for single-item transfers (see the "Single-item Data Transfers" section). The basic difference is in the order of operations and the extra steps to provide widget selection and confirmation before the selected operation proceeds to completion. You might implement a widget-based data transfer according to the following processing model.

1. The user chooses the Cut, Copy, or Paste option from a multiple-item transfer menu.

2. On the CHOOSE event for the chosen transfer option, the procedure enables all eligible widgets for selection (SELECTABLE attribute = TRUE) and makes all other nonparticipating widgets insensitive (SENSITIVE attribute = FALSE).

3. The user can now only select widgets for the selected data transfer option and confirm the selection.

4. When the user is finished selecting widgets, they invoke an option (for example, a chooseable button or menu item) that confirms and allows the operation to proceed to completion. (The user might also invoke an option to cancel the current operation and return to other application functions.)

5. On confirmation of widget selection, the procedure:

    a. Sets the appropriate CLIPBOARD handle attributes for the selected operation. This requires setting the MULTIPLE attribute to TRUE, and for Cut or Copy operations (write transfers) setting the ITEMS-PER-ROW attribute to format the data into lines of tab-separated items.

    b. Iterates through the widget list assigning each selected widget's SCREEN-VALUE attribute to the CLIPBOARD handle VALUE attribute for a Cut/Copy operation, or assigning the VALUE attribute to each SCREEN-VALUE attribute for a Paste operation. (You can implement the iteration through the widget list using either the NEXT/PREVIOUS-SIBLING or NEXT/PREVIOUS-TAB-ITEM attribute to return and save the handle of each succeeding widget in the list.)

    c. Completes the operation by setting the MULTIPLE attribute to FALSE and disabling selection and enabling sensitivity for all widgets (SELECTABLE = FALSE and SENSITIVE = TRUE).

6. The user can now perform other application functions.

In this implementation, the essential tasks in the multiple-item transfer are included in Item 5. Of course, there are many variations of this process that you can implement, such as providing preselected widget lists from which the user can choose (eliminating the need for the selection mode enabled in Item 2). This latter approach can employ a processing model similar to that used for data-based transfers.

3.3.2 Data-based Transfers

A typical data-based transfer differs from a widget-based transfer in that you directly reference the fields in a database (the record buffer) rather than the widgets visible on the display (the screen buffer). You might implement this type of transfer according to the following processing model.

1. The user chooses tables and fields from the database to participate in the selected operation.

2. The user indicates the operation (Cut, Copy, or Paste) to perform.

3. The procedure sets the CLIPBOARD handle MULTIPLE attribute to TRUE, and for Cut or Copy (write) operations, sets the ITEMS-PER-ROW attribute to the number of fields in each record participating in the transfer.

4. For a Cut/Copy operation, the procedure assigns each field to the VALUE attribute for each (selected) record in the table. For a Paste operation, the procedure assigns the VALUE attribute to the fields in each record created or updated in the table.

5. The procedure sets the MULTIPLE attribute to FALSE, ending the operation.

Note that for Cut/Copy operations on database fields you must access any noncharacter fields using the STRING function. For Paste (read) operations, your procedure must depend on the user to provide appropriately formatted data items in the clipboard. Your database validation functions can help to catch and respond to bad data in the clipboard.

3.3.3 Multiple-item Transfer Example

The following procedure demonstrates the essential elements of a multiple-item data transfer. It implements a basic data-based Copy operation using the customer table of the sports database. You can test the result by running the procedure and pasting the result into a window system editor like Notepad or Wordpad in Windows:

e-clpmul.p

```
CLIPBOARD:MULTIPLE = TRUE.
CLIPBOARD:ITEMS-PER-ROW = 11.

FOR EACH Customer:
    CLIPBOARD:VALUE = STRING(Cust-Num).
    CLIPBOARD:VALUE = Name.
    CLIPBOARD:VALUE = Address.
    CLIPBOARD:VALUE = Address2.
    CLIPBOARD:VALUE = City.
    CLIPBOARD:VALUE = State.
    CLIPBOARD:VALUE = Postal-Code.
    CLIPBOARD:VALUE = Country.
    CLIPBOARD:VALUE = Phone.
    CLIPBOARD:VALUE = STRING(Balance).
    CLIPBOARD:VALUE = STRING(Credit-Limit).
END.

CLIPBOARD:MULTIPLE = FALSE.
```

4

Named Pipes

In the UNIX and Windows NT environments, you can establish interprocess communications (IPC) between a non-Progress application (such as a C program or commercial software package) and a Progress session using named pipes. This facility provides a capability similar to Dynamic Data Exchange (DDE) in Windows, though it works very differently.

From the Progress 4GL, named pipes look and act like operating system files. To exchange data, the Progress application reads or writes to a named pipe, just as it does to a file. However, instead of a file at the end of the pipe, the non-Progress application reads or writes data to the Progress application.

This chapter contains the following sections:

4.1 Overview Of Named Pipes With Progress

Named pipes provide a general exchange mechanism for text data, and there is no practical limit to the types of data you can exchange using them. Any data you can access as a character string within Progress, you can read or write to a named pipe. For example, you can use named pipes to issue SQL requests to a Progress session from within a spreadsheet program and receive the resulting data in the spreadsheet. As such, you can use named pipes to implement some of the capabilities provided by Progress Host Language Call Interface (HLC) and Embedded SQL/C (ESQL/C).

Figure 4–1 shows a typical named pipe scenario.

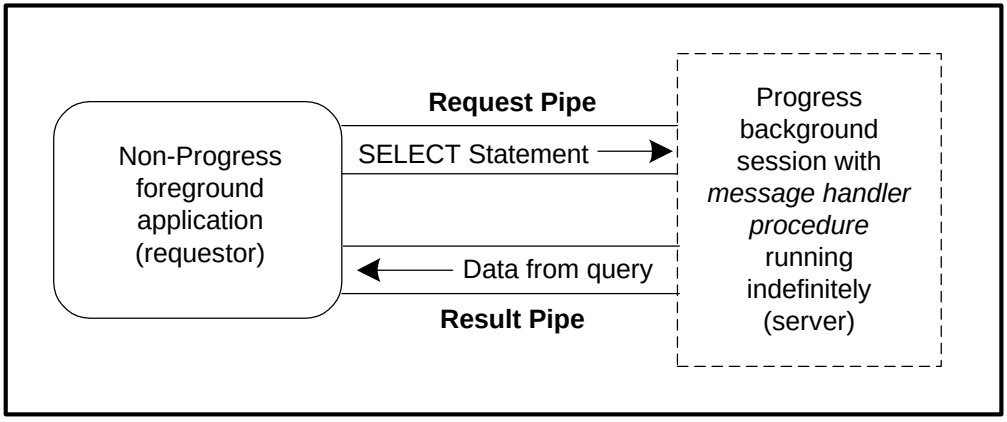


Figure 4–1: Typical Named Pipe Scenario

The *message handler procedure* acts as a server for a non-Progress application acting as requestor. The server reads each incoming request, processes it, and returns the results through a second named pipe. The messages can contain SQL statements, Progress statements, procedure names, or anything that your message handler procedure can manage.

Access From Progress

Progress accesses named pipes already created on your system; Progress does not create named pipes itself. Progress treats a named pipe the same way as it treats a text file. The Progress 4GL statements INPUT FROM, OUTPUT TO, DISPLAY, SET, EXPORT, and IMPORT all access named pipes and files identically.

Named Pipes and Files

Named pipes combine the features of files and unnamed pipes. Like a file, a named pipe has a name and any process with appropriate permissions can open it to read or write. Thus, unrelated processes can communicate over a named pipe. Like an unnamed pipe, a named pipe behaves like a first in/first out (FIFO) queue. A reading process reads and removes from the pipe the first unit of data written to the pipe that has not been read.

Uses For Named Pipes

The scenario in Figure 4–1 illustrates important core concepts, but it is a relatively simple example of what you can do with named pipes. For example, you can design your message handler procedure to handle requests from more than one non-Progress application user at a time. Another idea is to design a message handler that manages multi-line requests as well as single-line requests. This makes it possible for the requests to include, for example, Progress FOR EACH blocks.

4.1.1 Operational Characteristics Of Named Pipes

Once opened, named pipes act more like unnamed pipes than files. Data written to the named pipe is read in FIFO order. Once data written to a named pipe is read, it is removed from the named pipe. Also, the operating system regards individual reads and writes as unbreakable (atomic) units and issues them one at a time, unless the amount read or written exceeds the capacity of the named pipe. The capacity of a named pipe is the same as the capacity of an unnamed pipe. (The capacity of an unnamed pipe depends on the implementation; in UNIX environments, however, the amount is always 4,096 bytes or greater).

I/O Synchronization

Progress accesses named pipes using unbuffered I/O. This means that processes that read from and write to the same named pipe synchronize their reads and writes with each other. In Progress, this is true for opening as well as reading and writing named pipes. When a Progress process opens a named pipe for input, it blocks (waits) until another process opens the same named pipe for output. The reverse is also true—when a Progress process opens a named pipe for output, it blocks until another process opens the same named pipe for input.

When a process writes to a named pipe, the process blocks until another process reads from the named pipe. Similarly, when a process attempts to read from a named pipe, but there is nothing to read, the process blocks until something is written to the named pipe.

Message Interleaving

If multiple processes write messages to the same named pipe, the messages might be interleaved (mixed). However, as stated earlier, individual message reads and writes are atomic.

For example, suppose there are two processes, Process A and Process B. Each process writes several messages to the same named pipe. As they are written, some of the messages from Process A might become mixed with messages from Process B. However, an individual message cannot be interrupted by another message, since the messages are atomic. Figure 4–2 illustrates this example.

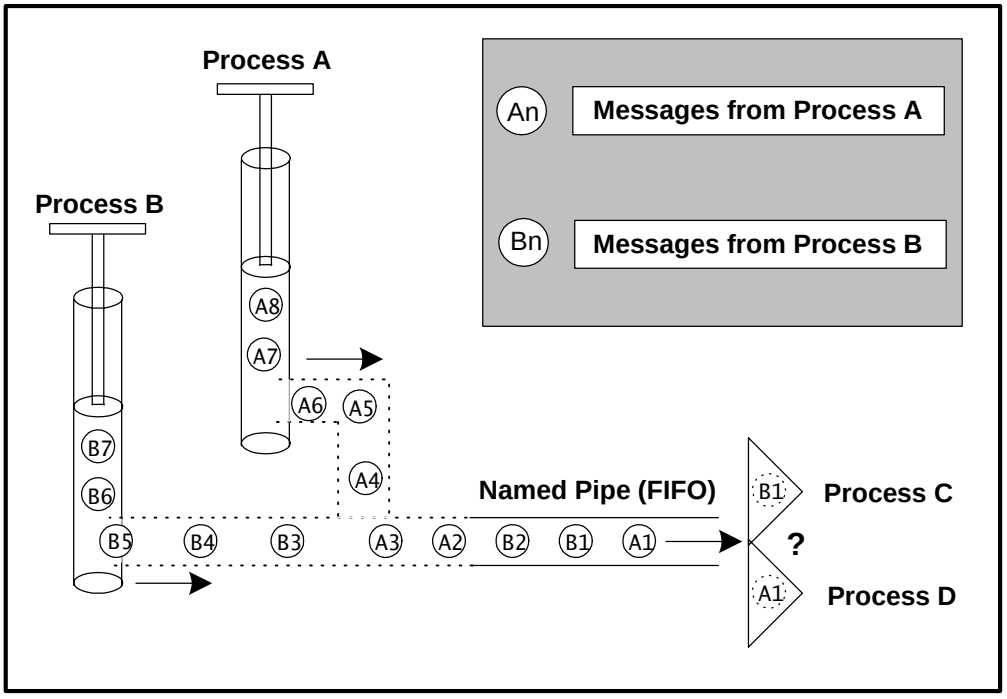


Figure 4–2: Writing Messages To a Named Pipe

Also, note that if two processes (Process C and Process D) simultaneously read from the same named pipe, they receive messages from both Process A and Process B in order of transmission, but whether Process C or D receives a particular message might be uncertain. In Figure 4–2, either Process C or Process D can receive message A1 or B1, if the processes read the pipe at the same time. The actual messages received by which process depend on the state of the system at the time of input. In other words, all applications that use a named pipe must establish a mutual protocol for effective cooperation.

4.1.2 Advantages and Disadvantages Of Named Pipes

A major advantage of using named pipes is that they provide a useful way to send one-line requests to a Progress background session running a message handler procedure. Multiple users can send requests through the same named pipe and each request is removed from the pipe as it is received. In addition, the message handler procedure can loop indefinitely looking for input because it blocks (waits) until there is something to read. Finally, output through named pipes is more efficient than writing a complete response to an ordinary file, closing the file, and then informing the recipient that the results are available. The receiving process can read the result through a named pipe as soon as it is written.

A disadvantage of named pipes is that multiple processes cannot use a single named pipe to send or receive multi-line messages, unless you define a more complex protocol to control message interleaving. Also, although synchronizing named pipe input and output is helpful in some situations, it is a problem in others. For example, if the message handler procedure running in the background Progress session starts returning results to an output named pipe, and for some reason the requestor is not ready to read the results, the message handler cannot move on to read the next request.

4.2 UNIX Named Pipes

This chapter provides information to get you started using named pipes with Progress on a UNIX system. You can find more information on UNIX named pipes in any of the books on advanced UNIX programming available in the public domain.

To use UNIX named pipes with Progress, follow these steps:

1 ♦ Create the named pipes using the UNIX `mknod` command in the command line or the `mknod()` system call from within C.

2 ♦ Start a Progress session in the background (Progress batch mode) running a message handler procedure. The message handler procedure runs indefinitely, searching for input from one named pipe, running requests, and shipping output through a second named pipe. (You must supply the message handler procedure. For a sample message handler procedure, see the "UNIX Named Pipe Examples" section.)

3 ♦ Run your non-Progress application. From within the application, issue messages through the first named pipe in Step 1 to the background Progress session, and receive replies through the second named pipe.

Once you create a named pipe, you can access it as if it were a text file. Thus, the only requirement for a non-Progress application to communicate with Progress via named pipes is that the application be able to write to and read from text files. Also, it is helpful for the application to have facilities for processing returned results (for example, string handling functions, buffers, etc.).

The following sections describe how to:

- Create a UNIX named pipe
- Delete a UNIX named pipe
- Use a UNIX named pipe between Progress and non-Progress applications

NOTE: Named pipes might be implemented on your system differently than described below.

4.2.1 Creating a Named Pipe

To create a UNIX named pipe, use the `mknod` command on the command line or the `mknod()` system call from a C program. The two techniques produce the same results. The examples in this chapter use the command-line technique.

Once you create a named pipe, its characteristics are similar to an ordinary file. For example, it is located in a directory, has a pathname, and exists until you delete it.

The `mknod` command has more than one form. This is the syntax for the form that creates a named pipe:

SYNTAX

```
mknod named-pipe-identifier p
```

The *named-pipe-identifier* is the pathname of the named pipe you want to create.

For example, to create a named pipe called `mypipe` in the current directory, type the following command:

```
mknod mypipe p
```

The following C function shows how to use the mknod() system call to create a named pipe:

```
int mkfifo(path) /* make FIFO */
char *path;
{
  return(mknod(path, S_IFIFO | 0666, 0));
}
```

For more information on mknod or mknod(), see your UNIX system documentation.

4.2.2 Deleting a Named Pipe

To delete a named pipe on UNIX, use the rm command.

For example, to delete the named pipe mypipe in the working directory, type the following command:

```
rm mypipe
```

From within a C program, use the unlink() system call. For more information on unlink(), see your system documentation.

4.2.3 Accessing a Named Pipe Within Progress

To access a named pipe from Progress, open it for input or output using the INPUT FROM and OUTPUT TO statements. For example, the following line of Progress code opens the previously created named pipe inpipe for input:

```
INPUT FROM inpipe NO-ECHO.
```

After invoking this statement, all input statements that use the unnamed stream, such as SET or IMPORT, take their input from inpipe.

4.2.4 UNIX Named Pipe Examples

The following examples show different uses of named pipes. To provide a simple example of how named pipes operate, the first example shows how to use the shell to create a named pipe, send a message to it, and read the message back. The second example shows how to use named pipes with Progress.

Example 1 — Creating and Using a Named Pipe From the Shell

In this example, the `cat` command sets up a message handler routine and the `echo` command acts as the requestor:

e-pipex1

```
# Named Pipe Example 1.
#
# Create named pipe...
mknod trypipe p
# Open named pipe and read message...
cat trypipe &
# Write message...
echo "This is a message!" > trypipe
# Delete pipe...
rm trypipe
```

To try this example, run the shell script, `e-pipex1`.

This script performs the following actions:

1. The `mknod` command creates a named pipe called `trypipe`.
2. The `cat` command opens `trypipe` for reading. It blocks because `trypipe` has not yet been opened for writing. Notice that an ampersand (`&`) is present at the end of the `cat` command; this runs the `cat` command as a background process.
3. The `echo` command opens `trypipe` for writing and writes a message. The `cat` command, blocked until now, resumes execution, and the message appears on the display.
4. The `rm` command deletes `trypipe`.

Example 2 — Using a Named Pipe With Progress

Before working with the following procedures, create a copy of the `demo` database with the PRODB utility:

```
prodb demo demo
```

This example shows a simple user program that sends one line requests to a Progress message handler routine running in the background, and displays the results. The example consists of four files:

- A script called `e-pipex2` that runs the example.
- The message handler procedure, `e-pipex2.p`.
- A subprocedure, `e-do-sql.p`.
- A C source file called `e-asksql.c` that implements the requestor

A display of these files follows:

e-pipex2

```
# Named Pipe Example 2.
#
# Create named pipes...
mknod inpipe p
mknod outpipe p
# Start Progress background session with e-pipex2.p running...
bpro demo -1 -p e-pipex2.p
# Run executable e-asksql...
e-asksql
# Terminate Progress background session...
echo "outpipe \"quit\"" > inpipe
cat outpipe
# Delete named pipes...
rm inpipe
rm outpipe
```

e-pipex2.p

```
DEF VAR sql-stmt AS CHAR FORMAT "x(220)".  /* Target variable for the request*/
DEF VAR out-pipe AS CHAR FORMAT "x(32)".  /* Holds the output file or FIFO */

REPEAT:                /* Do forever: */
  INPUT FROM inpipe NO-ECHO.  /* Set up to read from in-FIFO
                          named "inpipe". */
   REPEAT:              /* For each request received: */
    IMPORT out-pipe sql-stmt.  /* Get the output name
                          and the request. */
    OUTPUT TO VALUE(out-pipe) APPEND.  /* Set up to write results. */
    RUN e-do-sql.p sql-stmt.  /* Pass SQL request to sub-proc. */
    OUTPUT CLOSE.
  END.                 /* This loop ends when the in-FIFO
                          is empty. Just reopen it and */
END.                   /*  wait for the next request. */
```

e-do-sql.p

```
/* This program consists of a single line of code. */

{1}
```

e-asksql.c *(1 of 2)*

```
#include <stdio.h>
#include <fcntl.h>

main()

{
#define  LEN     250
         char   result[LEN];
         int     fdi, fdo, nread;
         char   request[LEN+8]; /* 8 for "outpipe " + punctuation */
         char   *ptr;
         int     validq, i;

         fdi = open("inpipe", O_WRONLY);
         if (fdi < 0)
         { printf("Error on inpipe open\n"); exit(1);}

         strcpy(request, "outpipe \""); /* request starts with 'outpipe "' */

         while (1)
         {
           printf("\n\nEnter your request (type [RETURN] to exit):\n");
           ptr = request+9;
           nread = read(0, ptr, LEN);
           if (nread < 2)
               exit(0);
           else
           {
             validq = 1;             /* valid query? */
             for (i = 9; i<nread+9; i++)
                 if (request[i] == '\"')
                 { printf("Use only single quotes in queries.\n");
                   validq = 0;
                   break;
                 }
             if (! validq) continue;
             ptr += nread-1;
             *ptr++ = '\"';
             *ptr++ = '\n';
             *ptr++ = '\0';
             write(fdi, request, strlen(request));
```

e-asksql.c *(2 of 2)*

```
            sleep(1);
            fdo = open("outpipe", O_RDONLY);
            if (fdo < 0)
            { printf("Error on outpipe open\n"); exit(1);}

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

To prepare and run the example, follow these steps:

1 ♦ Use `cc` to compile and link the requestor source `e-asksql.c` to produce the executable `e-asksql`, as shown:

```
cc e-asksql.c -o e-asksql
```

2 ♦ Execute the `e-pipex2` script to run the example:

```
e-pipex2
```

The e-pipex2 script performs the following actions:

1. It uses mknod to create two named pipes: inpipe and outpipe. Named pipe inpipe carries requests from the requestor to the message handler routine. Named pipe outpipe carries results back in the opposite direction, from the message handler to the requestor.

   Figure 4–3 illustrates this process.

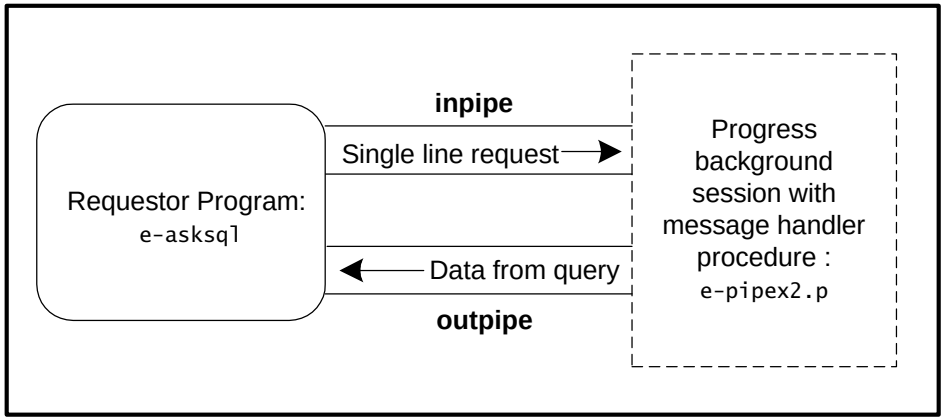


Figure 4–3: Named Pipes and Progress

2. It starts a background Progress session that runs the message handler procedure, e-pipex2.p , with the demo database. The following line in e-pipex2.p opens the named pipe inpipe for input:

```
INPUT FROM inpipe NO-ECHO.
```

   Notice that Progress accesses named pipes in exactly the same way as UNIX text files. At this point, e-pipex2.p blocks until a requestor process opens inpipe for output.

3. After starting the Progress message handler, the script starts the requestor, e-asksql, which opens inpipe for output using the following statements:

```
int fdi, fdo, nread;
            .
            .
            .
fdi = open("inpipe", O_WRONLY);
```

4. As `e-asksql` opens `inpipe`, `e-pipex2.p` unblocks and blocks again as it attempts to read a message from `inpipe`. The message handler procedure, `e-pipex2.p`, expects single-line requests from requestors, and can handle more than one requestor. This is the syntax for message handler requests:

   SYNTAX

   ```
   output-pipe-name SQL-statement
   ```

   Each request contains the name of the named (*output-named-pipe*) pipe from which the requestor expects to receive results and an SQL statement (*SQL-statement*) surrounded by double quotes (" "). The message handler procedure reads these messages with the following statement:

   ```
   IMPORT out-pipe sql-stmt.
   ```

   Each requestor must specify a unique value for *output-pipe-name*, or results might be intermixed. (Using the requestor's PID number as part of the name ensures uniqueness. However, for that to work, the requestor probably has to create its own named pipe using the `mknod()` system call.) Note that for this example, the requestor, `e-asksql`, uses the existing named pipe `outpipe` created by the script.

5. As the message handler waits for input, the requestor displays the following prompt:

   ```
   Enter your request (type [RETURN] to exit):
   ```

   You can enter a one-line SQL query like the following SELECT from the demo database:

   ```
   SELECT name FROM customer.
   ```

6. The requestor constructs a message from the name of the output pipe (`outpipe`, in the example) and the contents of your query, and writes the message to `inpipe`, as in the following statements from `e-asksql.c`:

```
char request[LEN+8]; /* 8 for "outpipe " + punctuation */
                .
                .
                .
write(fdi, request, strlen(request));
```

7. As the message handler receives (and removes) the message from `inpipe`, it unblocks and opens the output pipe named in the message with the following statement:

```
OUTPUT TO VALUE(out-pipe).
```

8. The message handler blocks again, waiting for the requestor to open the same pipe for input (to receive the query result), as in the following statements from `e-asksql.c`:

```
int fdi, fdo, nread;
            .
            .
            .
fdo = open("outpipe", O_RDONLY);
```

9. The Progress message handler then continues to compile and run the SQL query using the following statement:

```
RUN e-do-sql.p sql-stmt.
```

As the query generates output, Progress writes it one line at a time to the named pipe specified by `outpipe`. The requestor reads each line as it is written to the output pipe, as in the following statements from `e-asksql`. In the example, the requestor also writes each line to its standard output:

NOTE: If there is no output, you might have entered your SQL statement incorrectly. This causes `e-pipex2.p` to terminate. To trap this type of error, write the SQL statement to a file instead of to a named pipe, then compile the file. If the compilation is successful, run it.

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

Note that although the query procedure, `e-do-sql.p`, contains only the single procedure parameter, `{1}`, you can extend it with formatting statements to avoid having to include these in each query, as in the following examples:

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

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

10. The example requestor, `e-asksql`, continues to prompt for queries, repeating Actions 5 through 9, until you press **RETURN** with no additional input.

11. After the requestor terminates, the `e-pipex2` script terminates the Progress background process with the following commands:

```
echo "outpipe \"quit\"" > inpipe
cat outpipe
```

The first command sends the Progress QUIT statement to the message handler (instead of an SQL statement). The second command takes the place of Action 8, originally handled by the requestor. The requestor does not send the QUIT to terminate the Progress background process so that multiple copies of the requestor — each with its own output pipe — can run without affecting the message handler. It is necessary because a process blocks until a named pipe it opens for writing is opened for reading (see the "Operational Characteristics Of Named Pipes" section). In this case, the message handler opens named pipe `outpipe` for writing, and cannot execute QUIT until the `cat` command opens `outpipe` for reading.

12. The, `e-pipex2` script uses the `rm` command to remove the named pipes that it created.

4.3 Windows NT Named Pipes

Progress Versions 8.0B and higher support named pipes in the Windows NT environment. In general, named pipes in the Windows NT environment behave similarly to UNIX named pipes. Some differences are:

- You cannot create Windows NT named pipes from the command line
- Windows NT named pipes have different C language interfaces

4.3.1 Accessing Windows NT Named Pipes

To access a Windows NT named pipe, you create it, connect it, read it, write it, and close it. Table 4–1 lists these tasks and their C and 4GL equivalents.

Table 4–1: Using C and 4GL To Access Windows NT Named Pipes

Task	C	4GL
Create	CreateNamedPipe()	None
Connect	ConnectNamedPipe()	None
Read	ReadFile() FlushFileBuffers()	INPUT FROM
Write	WriteFile()	OUTPUT TO
Close	CloseHandle()	INPUT CLOSE OUTPUT CLOSE

As Table 4–1 shows, C lets you create, connect, read, write, and close Windows NT named pipes, while the 4GL lets you read, write, and close them.

Actually, the 4GL lets you perform all the tasks in the table if you use the 4GL's access to DLLs to call into `kernel32.dll`, which contains all the C functions in the table. For more information on using the 4GL to access DLLs, see Chapter 5, "Shared Library and DLL Support."

4.3.2 Linking Progress and non-Progress Processes Using Windows NT Named Pipes

You can link Progress and non-Progress processes using Windows NT named pipes, just as you can using UNIX named pipes. The resulting application consists of:

- A 32-bit Progress Windows NT client running a 4GL application in the background.
- A non-Progress program.

The Progress program reads, writes, and closes one or more Windows NT named pipes. The non-Progress program creates, connects, reads, writes, and closes the named pipe or pipes.

4.3.3 Building and Running the Sample Windows NT Named Pipes Application

This update describes a sample application consisting of a 4GL program and a C program that communicate through a Windows NT named pipe. Both programs can read and write the named pipe. The 4GL program accesses the sports database. When you run the application, you must tell one program to write the named pipe and the other program to read it.

If you tell the C program to write and the 4GL program to read,

The C program...	While the 4GL program...
• Solicits a customer number from the user. • Writes the customer number to the named pipe.	• Reads the named pipe, getting the customer number the user entered. • Retrieves the row of the customer with the specified customer number. • Displays the row of the customer table.

If you tell the 4GL program to write and the C program to read,

The 4GL Program...	While the C program...
• Reads the entire customer table. • Writes the name of each customer to the named pipe.	• Reads the named pipe into a buffer. • Displays the number of bytes read and the contents of the buffer.

4.3.4 Coding the 4GL Program

The 4GL program:

- Assumes that the C program creates and connects the named pipe.
- Refers to the named pipe using the name the C program specifies.
- Uses the INPUT FROM statement to read the named pipe.
- Uses the OUTPUT TO statements to write the named pipe.
- Uses the INPUT CLOSE and OUTPUT CLOSE statements to close the named pipe.

The following 4GL program demonstrates reading and writing a Windows NT named pipe "custpipe:"

e-4glpip.p *(1 of 2)*

```
/* e-4glpip.p */
/* 4GL program that reads and writes a Windows NT named pipes */

/* 1. Define buttons */
DEF BUTTON bWrite LABEL "Write to Pipe".
DEF BUTTON bRead  LABEL "Read from Pipe".
DEF BUTTON bQuit  LABEL "Quit".

/* 2. Define form */
FORM SKIP(5)
  SPACE(5) bWrite SPACE(5) bRead SPACE(5)
  SKIP(1)
  SPACE(18) bQuit
  SKIP(5)
WITH FRAME f TITLE "Pipe Test".

/* 3. Define write trigger */
ON CHOOSE OF bWrite IN FRAME f
DO:
  OUTPUT TO \\.\pipe\custpipe APPEND.
  FOR EACH customer:
    DISPLAY name.
  END.
  OUTPUT CLOSE.
END.
```

e-4glpip.p *(2 of 2)*

```
/* 4. Define read trigger */
ON CHOOSE OF bRead IN FRAME f
DO:
  DEF VARIABLE i AS INTEGER.
  INPUT FROM \\.\pipe\custpipe.
  SET i.
  INPUT CLOSE.
  FIND customer WHERE cust-num = i.
  DISPLAY customer WITH 2 COLUMNS FRAME y OVERLAY TITLE "Customer Info".
END.

/* 5. Define quit trigger */
ON CHOOSE OF bQuit IN FRAME f
DO:
  APPLY "window-close" TO CURRENT-WINDOW.
END.

/* 6. Enable all objects, then wait on a close event */
ENABLE ALL WITH FRAME f.
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

1. The program defines three buttons, labeled “Write to Pipe,” “Read from Pipe,” and “Quit.”

2. The program defines a form to contain the buttons.

3. The program defines a trigger for the Write to Pipe button. The trigger redirects output to named pipe custpipe, displays (to named pipe custpipe) the name of each customer in the Sports database, and closes the named pipe.

   In an OUTPUT TO statement, `\\.` means the current machine. To communicate with remote machine “pcdev68,” for example, use `\\pcdev68`. This follows Uniform Naming Conventions (UNC).

   The OUTPUT TO statement uses the APPEND option, which causes Progress to open the named pipe without first creating it. This is necessary because Progress 4GL cannot create named pipes.

4. The program defines a trigger for the Read to Pipe button. The trigger defines an integer data item, reads named pipe custpipe, assigns the value read (a customer number) to the integer data item, closes the named pipe, retrieves the row of the customer table with the specified customer number, and displays the columns of the row.

   The INPUT FROM statement assumes that the pipe exists and that another process writes to it. The INPUT FROM statement blocks until the other process writes to the named pipe.

5. The program defines a trigger for the Quit button.

6. The program enables all objects in the frame and waits on a close event.

4.3.5 Coding the C Program

The non-Progress program creates, connects, reads, writes, and closes the Windows NT named pipe or pipes.

The following C program demonstrates creating, connecting, reading, writing, and closing Windows NT named pipe "custpipe:"

e-cpipe.c *(1 of 3)*

```
/* C program that reads and writes a Windows NT named pipe */
/* 1. Declare include files */
#include <windows.h>
#include <stdio.h>
#include <wincon.h>
#include <winerror.h>
#include <conio.h>

void main()
{
/* 2. Define automatic data items */
  HANDLE hPipe;
  char buffer[4096];
  DWORD dwBytesRead;
  BOOL bRet;
  int iPipeType;
  int iLen;
```

e-cpipe.c *(2 of 3)*

```
/* 3. Make window title meaningful */
/* 4. Create the named pipe */
  printf("Creating Named Pipe custpipe\n");
  hPipe = CreateNamedPipe("\\.\pipe\custpipe",
    PIPE_ACCESS_DUPLEX,
    PIPE_TYPE_BYTE | PIPE_READMODE_BYTE | PIPE_WAIT,
    1,
    0,
    0,
    NMPWAIT_USE_DEFAULT_WAIT,
    NULL);
  if (hPipe == INVALID_HANDLE_VALUE)
  {
    printf("Error creating pipe, %ld\n", GetLastError());
    exit(GetLastError());
  }
  printf("custpipe created successfully\n");
/* 5. Solicit user input */
  do
  { printf("\nPress 1 for Read, 2 for Write:  ");
    iPipeType = getch(); }
  while (iPipeType != '1' && iPipeType != '2');
/* 6. Connect the named pipe */
  printf("\nWaiting for connection...");
  if (ConnectNamedPipe(hPipe, NULL) == FALSE)
  {
    printf("Error connecting to named pipe, %'d", GetLastError());
    exit(GetLastError());
  }
  printf("and connection established\n");
/* 7. Read and write the pipe */
  switch(iPipeType)
    {
    case '1': /* read */
      while (TRUE)
      {
        bRet = ReadFile(hPipe, &buffer, 4096, &dwBytesRead, NULL);
        if (bRet == FALSE)
        {
          /* if the pipe went away, the 4GL app did an OUTPUT CLOSE */
          if (GetLastError() == ERROR_BROKEN_PIPE)
          printf("Error reading from pipe, %ld\n", GetLastError());
          break;
        }
```

e-cpipe.c *(3 of 3)*

```
        printf("Data read = %ld %s\n", dwBytesTead, buffer);
      } /* end while */
      break;
    } /* end switch */
    case '2': /* write */
      while (TRUE)
        {
          printf(
             "Enter data to send, followed by ENTER: (Blank to exit) \n"
            );
          gets(buffer);
          if (buffer[0] == 0)
            break;
          iLen = strlen(buffer);
          buffer[iLen++] = 0x0d; /* CR */
          buffer[iLen++] = 0x0a; /* LF */
          bRet = WriteFile(hPipe, &buffer, (DWORD) iLen, &dwBytesRead,
                      NULL);
          if (bRet)
          {
            FlushFileBuffers(hPipe);
            printf("%ld bytes flushed to pipe\n", dwBytesRead);
          }
          else
            printf("Error writing to pipe: %d\n", GetLastError());
        } /* end while */
        break;
      } /* end switch */
/* 8. Close and exit */
  CloseHandle(hPipe);
  printf("custpipe closed successfully\nBye");
}
```

1. The program declares include files.

2. The program defines data items.

3. The program calls an NT Console API function to give the window a more descriptive title.

4. The program creates named pipe custpipe. The PIPE_ACCESS_DUPLEX flag makes the pipe read/write. The PIPE_WAIT flag makes the pipe synchronous. The program checks for errors and prints debugging messages here and throughout.

5. The program solicits and accepts a value (“1" to read the pipe, “2" to write the pipe) from the user.

6. The program connects the named pipe. This makes the pipe available to other applications, processes and threads.

7. The program either reads the pipe and displays the number of bytes read along with the actual data, or else solicits a customer number, appends a carriage return and a line feed (CR/LF), writes the result to the named pipe, and flushes the buffers.

   The program uses the same API calls for named pipes as for files. Named pipes are an integral part of the Windows NT file system, just as they are an integral part of UNIX file systems.

   If the program did not append a CR/LF to the data it writes to the named pipe, the 4GL program's SET input statement would wait for a line terminator or EOF marker, which does not ordinarily appear until the pipe is closed.

   The two programs use a quick and dirty hack to signal "named pipe EOF." The 4GL program closes the named pipe with the OUTPUT CLOSE statement, which causes the C program's ReadFile() call to raise a BROKEN_PIPE error, which the C program interprets as "named pipe EOF."

8. The program closes the named pipe and exits.

4.3.6 Running the Application

Before you run the application, you must compile and link the C program.

This procedure assumes that the name of the C executable is `e-cpipe`.

Follow these steps to run the application:

1 ♦ In Windows NT, open (or go to an existing) command window, which resembles an MS-DOS box.

2 ♦ Enter `prowin32 -p e-4glpip.p -1 sports`.

   – The Progress process starts.

3 ♦ Enter `e-cpipe`.

   – The non-Progress process starts.

4 ♦ In the C program, enter “1" to select reading a named pipe.

5 ♦ In the 4GL program, select “Write to Pipe.”

- The 4GL program writes the customer name in each row of the customer table to the named pipe.
- The C program reads the named pipe and displays the customer names.
- The programs terminate.

You can rerun the application. In the C program, enter “2" to select writing a named pipe, then enter a customer number. In the 4GL program, select “Read from Pipe.” The C program writes the customer number to the named pipe. The 4GL program reads the named pipe, retrieves the row of the customer table with the specified customer number, and displays the row’s columns.

5

Shared Library and DLL Support

A *shared library* is a file that contains a collection of compiled functions (routines) that can be accessed by applications. Such a file is called a *shared object* or *shared library* on UNIX and a *dynamic link library* (DLL) on Windows.

An application links to these routines at run time rather than at build time, and shares the code with other applications that link to them. Thus, shared libraries promote code reuse (because an application can reference third-party routines) and upgradeability (because any enhancement to a shared library becomes immediately available to your application, without rebuilding).

Progress lets you link and execute shared library routines from a 4GL procedure. Using these routines, you can write Progress applications that perform a wide range of third-party functions from graphics to advanced multi-media (sound and video) production.

For more information on shared library concepts and facilities, see the documentation for your operating system.

The following sections describe how to use shared libraries in your Progress applications:

5.1 Using Shared Libraries

Progress lets you access shared libraries by declaring and executing the shared library routines in a manner similar to Progress internal procedures. Thus, to use a shared library routine from within Progress, you must complete the following tasks:

- Declare each routine, including input and output parameter definitions, in a PROCEDURE statement.
- For each shared library parameter definition, specify the parameter data type that matches the C data type of the corresponding routine parameter.
- For each Progress variable you pass as a routine parameter, ensure that the data type of the variable is compatible with the parameter data type.
- Execute the shared library routine using the RUN statement. Pass the specified fields, variables, and any required Windows widget handles.
- Build and manage any structures used by the routine.

Progress also allows you to program how long shared libraries remain memory-resident during application execution.

The following sections describe how to perform these tasks.

5.2 Declaring a Shared Library Routine

To access a shared library entry point from within Progress, you must declare the routine using a PROCEDURE statement. The declaration is similar to an internal procedure declaration, but instead of containing procedure code, it contains options and parameter definitions that specify how to access the external shared library routine. This is the syntax for a shared library routine declaration:

SYNTAX

```
PROCEDURE proc-name  EXTERNAL "dllname"
  [ CDECL | PASCAL | STDCALL ]
  [ ORDINAL n ]
  [ PERSISTENT ] :
  [ parameter-definition ] ...
  END [ PROCEDURE ] .
```

NOTE: The PROCEDURE statement that declares the DLL routine can appear anywhere within the 4GL source file. It does **not** have to precede the RUN statement that invokes it. Typically, the procedure declarations appear at the end of the source file or are written in a separate file that is included in the source file.

5.2.1 Options For Shared Library Routine Declarations

In the syntax, the *proc-name* value is the name of your shared library routine. The EXTERNAL option indicates that the procedure being declared is an internal procedure implemented by an external shared library routine. The *dllname* value is the name of the shared library file that contains the routine.

If you specify the PERSISTENT option, the entire shared library remains loaded until Progress exits, or until you explicitly unload the shared library by using the 4GL RELEASE EXTERNAL statement.

Each *parameter-definition* value consists of a DEFINE PARAMETER statement. This is the syntax you must use for the DEFINE PARAMETER statement in a DLL routine declaration:

SYNTAX

```
DEFINE
  { INPUT | OUTPUT | INPUT-OUTPUT | RETURN }
  PARAMETER parameter
  {     { AS [ HANDLE [ TO ] ] datatype }
    |  { LIKE field }
  }
```

Each DEFINE PARAMETER statement specifies an INPUT, OUTPUT, or INPUT-OUTPUT parameter in the order it appears in the calling sequence of your shared library routine. You can also (optionally) define one RETURN parameter that provides the function return value of your shared library routine.

The parameter name (*parameter*) serves only as a place holder, and can take any unique identifier value. The AS *datatype* and LIKE *field* options require a special set of data types for shared library parameter definitions. These are described in the following section.

Windows DLL Calling Conventions

The PROCEDURE statement lets you specify the calling convention — C, Pascal, or standard—that your DLL requires. The default is the standard calling convention. Many Windows functions use the standard calling convention. Windows functions that take a variable number of arguments, such as `wsprintf()`, often use the C calling convention. For more information on the C, Pascal, and standard calling conventions, see the *Microsoft C Language Reference*. For more information on the calling convention that a particular Windows function requires, see the *Microsoft Windows Programmer's Reference*. Table 5–1 shows how to specify calling conventions.

Table 5–1: DLL Calling Conventions

To Specify This Calling Convention...	Code This Option...
C	C
Pascal	PASCAL
Standard	STDCALL

You can alternatively specify the DLL entry point by number with the ORDINAL *n* option, where *n* is the ordinal number of the entry point in the library. If you specify the entry point by number, *proc-name* can have any unreserved identifier value, but you must use the same value for the RUN statement that executes the DLL routine.

5.2.2 Shared Library Parameter Data Types

For shared library parameter definitions, Progress provides a special set of data types to match the standard C and Windows data types used in shared library calling sequences. These are the only data types you can specify for the AS *datatype* option in shared library parameter definitions:

- BYTE
- SHORT
- UNSIGNED-SHORT
- LONG
- FLOAT
- DOUBLE
- CHARACTER
- MEMPTR

CHARACTER and MEMPTR are standard Progress variable data types available for other uses. You can only use the remaining listed data types for shared library parameter definitions.

The MEMPTR data type specifies a pointer to a region of memory. It lets you define and pass C-compatible structures to shared library routines. It also provides a safe way to pass CHARACTER variables to shared library routines that modify them. For more information on using the MEMPTR data type, see the "Using Structure Parameters" section.

Data Type Compatibilities

For each parameter definition, you must specify a data type that is compatible with the standard C or Windows data type of the corresponding DLL routine parameter. Many data types referenced by DLL routines have the same memory size and usage.

Table 5–2 lists each supported memory size and usage, examples of corresponding C and Windows data types, and the Progress DLL parameter data type you must use for each one.

Table 5–2: C and DLL Parameter Data Type Compatibilities *(1 of 2)*

Data Type Size and Usage	Example C and Windows Data Types	DLL Parameter Data Type
8-bit unsigned integer	`char, BYTE`	BYTE
16-bit signed integer	`short, ATOM, BOOL`	SHORT
16-bit unsigned integer	`unsigned short`	UNSIGNED-SHORT
32-bit signed integer	`long, COLORREF, int`[1]	LONG
4-byte floating point	`float`	FLOAT
8-byte floating point	`double`	DOUBLE

Table 5–2: C and DLL Parameter Data Type Compatibilities *(2 of 2)*

Data Type Size and Usage	Example C and Windows Data Types	DLL Parameter Data Type
Address (usually 32 bits)	`char*`	CHARACTER
Address (usually 32 bits)	`double*`	HANDLE TO DOUBLE[2]
Address (usually 32 bits)	`char*, LPCSTR, LPWINDOWPOS`	MEMPTR

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

2 You can also use the HANDLE option to specify pointers to FLOAT, LONG, SHORT, and BYTE data types. For a CHARACTER parameter, it is redundant because this data type is always passed using a pointer (*char**).

For more information on C and Windows data types, see the documentation for the operating system(s) on which your applications will run.

Other Data Type Options

For shared library parameters that pass pointers to scalar values (for example, SHORT, DOUBLE, etc.), Progress provides the HANDLE option. You must use this option for INPUT parameters that require pointers to scalar values instead of the values themselves. Although Progress automatically passes pointers for OUTPUT and INPUT-OUTPUT parameters, the HANDLE option is recommended for clarity.

If you use the LIKE option to specify the data type of a parameter definition, *field* might only be a database field defined as CHARACTER or a Progress variable defined as CHARACTER or MEMPTR.

NOTE: Progress does not support database fields defined as MEMPTR.

5.3 Executing a Shared Library Routine

Use the 4GL RUN statement to execute shared library routines. This is the syntax for RUN statements that execute shared library routines:

SYNTAX

```
RUN proc-name [ ( parameter-list ) ]
```

You use this statement in virtually the same way as for internal Progress procedures or external procedure files. The only differences are that positional arguments referenced in procedure files by ordinal identifiers ({1}, {2}, etc.) have no meaning for shared library routines (which cannot compile them), and errors returned by shared library routines do not raise the Progress error condition. They might, however, return error messages.

5.3.1 Options For Shared Library Routine Execution

The *proc-name* in the RUN statement must match the *proc-name* in the corresponding PROCEDURE statement, whether or not you declare the shared library routine with the ORDINAL option. Also, *proc-name* must be declared in the same 4GL source file where the RUN statement is executed.

The *parameter-list* contains INPUT, OUTPUT, and INPUT-OUTPUT parameters in the order defined for the corresponding PROCEDURE statement. If you specify a RETURN parameter for the PROCEDURE statement, you must match it with an OUTPUT parameter in the corresponding RUN statement. You cannot specify the return value for a shared library function as a CHARACTER variable. You must pass character strings used as the return value to a MEMPTR variable. See the "Passing CHARACTER Values To Shared Library Routines" section for more information. The RUN statement parameter data types must match the data types of the corresponding parameter definitions for the PROCEDURE statement. Any mismatch causes a run-time error.

5.3.2 RUN Statement Parameter Data Types

Note that while shared library parameter definitions use a special set of data types, the corresponding expressions, fields, and variables passed in the RUN statement have standard Progress data types. You must ensure that your RUN statement parameters have data types that are compatible with their corresponding shared library parameter definitions. Table 5–3 lists each shared library parameter data type and the Progress data type that is compatible with it in a RUN statement.

Table 5–3: Shared Library and RUN Statement Parameter Compatibilities

Shared Library Parameter Data Types	RUN Parameter Data Types
BYTE	INTEGER
SHORT	INTEGER
UNSIGNED-SHORT	INTEGER
LONG	INTEGER
FLOAT	DECIMAL
DOUBLE	DECIMAL
CHARACTER	CHARACTER
MEMPTR	MEMPTR

CAUTION: For CHARACTER parameters, Progress always passes the routine a pointer to the character or character string value rather than the value itself. If the routine modifies the value, it can also modify Progress memory outside the bounds of the CHARACTER value, with unpredictable results. To avoid this kind of memory fault, pass the character string as a MEMPTR parameter instead. For more information, see the "Passing CHARACTER Values To Shared Library Routines" section.

5.3.3 Passing NULL Values

If you have to pass a NULL value to a shared library routine, do not pass a null MEMPTR variable. Instead, define the INPUT or INPUT-OUTPUT parameter as a LONG, and pass it 0 when you run the routine. If this conflicts with calling the shared library entry point another way, you can create a second declaration using the ORDINAL option of the PROCEDURE statement. (Note that this option is not available on UNIX.)

5.4 Using Structure Parameters

Many shared library routines require a pointer to a structure rather then a scalar value. Progress provides the MEMPTR DEFINE VARIABLE statement. Once initialized, you can use the MEMPTR variable with a set of 4GL statements and functions to build, read, and modify the contents of any C-compatible structure used by your shared library routines. You can also use the MEMPTR variable to safely pass CHARACTER values to shared library routines that modify them. For basic information on the MEMPTR data type and how to use it, see Chapter 1, “Introduction.” This section provides information on using MEMPTR variables as parameters to shared library routines.

NOTE: You **cannot** use the Progress RAW data type to build structures that you pass to shared library routines.

5.4.1 Initializing and Uninitializing MEMPTR Variables

Typically when you initialize a MEMPTR variable, you use the SET-SIZE statement to allocate a region of memory for a specified size and associate it with the variable. You can also use a shared library routine to allocate the memory for a MEMPTR variable by passing the variable appropriately as an OUTPUT parameter to the routine that returns a structure. Then, to complete MEMPTR initialization, you must use the SET-SIZE statement so that Progress knows how big the memory area is.

You **must** know the exact size of data returned by the shared library routine to initialize the MEMPTR variable properly with the SET-SIZE statement. If you use an incorrect value, you might not be able to access the data as you expect. Note also that if you do not complete initialization of a shared library pre-initialized MEMPTR variable using the SET-SIZE statement, Progress does not perform any bounds checking when you read or modify contents of the structure.

CAUTION: If you specify a size that is too small, Progress prevents you from accessing parts of the returned structure that lie outside the specified region. However, if you specify a size that is too large (or do not complete initialization at all), you might cause a memory violation by inappropriately accessing memory outside the area of the structure. This can result in loss of data. To determine the size of structures allocated and returned by shared library routines, see the documentation for the routine you are calling.

Checking the Size Of a MEMPTR Variable

For any MEMPTR variable, you can determine the size of the structure by using the GET-SIZE function. If SET-SIZE was never called, Progress will return a value of 0.

Freeing Memory Associated With a MEMPTR Variable

The region of memory associated with a MEMPTR variable remains allocated until it is freed. In some cases, the shared library routine frees the memory; in other cases, the calling procedure must free the memory using the SET-SIZE statement to set its size to zero (0). Progress cannot free the memory for you. It is up to you to ensure that the memory is freed, depending on the functionality of each shared library routine you use.

5.4.2 Passing CHARACTER Values To Shared Library Routines

If you are passing a Progress character string to a shared library routine, you can pass it as a CHARACTER variable or expression. However, if you expect the shared library routine to modify the value, you must set up and pass the Progress character string in a MEMPTR memory region as a null-terminated string. Otherwise, the shared library routine might inappropriately modify Progress memory outside the bounds of the CHARACTER value with unpredictable results. To return a character string from a shared library routine, the OUTPUT parameter must be a MEMPTR. You then use the GET-STRING function to extract the CHARACTER value.

The following code fragment shows how to repackage a CHARACTER value in a MEMPTR memory region, and return the new value from a DLL routine:

```
DEFINE VARIABLE MemptrVar AS MEMPTR.
DEFINE VARIABLE CharString AS CHARACTER.

PROCEDURE DLLfunction EXTERNAL anysystem.dll ORDINAL 10:
  DEFINE INPUT-OUTPUT PARAMETER StringParm AS MEMPTR.
END PROCEDURE.
                      .
                      .
                      .
SET-SIZE(MemptrVar) = LENGTH(CharString) + 1.
PUT-STRING(MemptrVar,1) = CharString.

RUN DLLfunction (INPUT-OUTPUT MemptrVar).

CharString = GET-STRING(MemptrVar).
                      .
                      .
                      .
```

The DLL routine is the tenth function in the `anysystem.dll` file. The SET-SIZE statement allocates to MemptrVar a memory region large enough to hold the CharString value plus a null terminator. The PUT-STRING statement builds a structure with one location containing the null-terminated value of CharString. After passing MemptrVar to the DLL routine, the GET-STRING statement returns the (new) value to CharString.

5.5 DLL Routines and Progress Widgets (Windows Only)

Some system DLL routines can manipulate windows on the display. To allow these routines to interact with Progress windows, every Progress user interface widget has the HWND attribute. This attribute contains the Windows handle to the window that contains the widget. You can pass this window handle as an integer value to a DLL routine using a LONG `DLL` parameter or a LONG location in a structure (MEMPTR) parameter.

CAUTION: When you pass an `HWND` to a DLL routine, that routine has complete control of a system window that Progress is using for its own widget management. This means that the DLL routine can modify the window's attributes and even destroy the window itself without Progress knowing about it. Such actions can cause unintended effects on your Progress application.

5.6 Loading and Unloading Shared Libraries

Progress provides several options for loading and unloading shared libraries. If you want Progress to automatically load and unload a shared library each time you invoke it, use the PROCEDURE statement without the PERSISTENT option. If you want Progress to automatically load a shared library and keep it loaded, use the PROCEDURE statement with the PERSISTENT option. To manually unload a shared library, use the 4GL RELEASE EXTERNAL statement.

5.7 Examples

The following procedure uses the Progress MESSAGE statement to display a message in an alert box. It then calls the `MessageBoxA` routine from `user32.dll` to display the same message in an identical alert box:

e-dllex1.p

```
/* e-dllex1.p */

DEFINE VARIABLE result AS INTEGER
  MESSAGE "   It's a whole new world!"
  VIEW-AS
    ALERT-BOX MESSAGE
    BUTTONS OK
    TITLE "Progress DLL Access".

RUN MessageBoxA (0, "   It's A Whole New World!",
  "Progress DLL Access - from the DLL!", 0, OUTPUT result).

PROCEDURE MessageBoxA EXTERNAL "user32.dll":
    DEFINE INPUT PARAMETER hwnd AS LONG.
    DEFINE INPUT PARAMETER mbtext AS CHARACTER.
    DEFINE INPUT PARAMETER mbtitle AS CHARACTER.
    DEFINE INPUT PARAMETER style AS LONG.
    DEFINE RETURN PARAMETER result AS LONG.
END.
```

The following procedure uses the `sndPlaySoundA` routine from `winmm.dll`. The procedure allows the user to select a sound to play and then invokes the DLL routine to play the sound. The DLL routine takes two input parameters and returns a status code:

e-dllex2.p

```
DEFINE VARIABLE wave-name AS CHARACTER INITIAL ? NO-UNDO.
DEFINE VARIABLE play-status AS INTEGER.

SYSTEM-DIALOG GET-FILE wave-name
  TITLE "Choose the Sound"
  FILTERS "Wave Files (*.wav)" "*.wav"
  MUST-EXIST USE-FILENAME.

RUN sndPlaySoundA (INPUT wave-name, INPUT 2,
                   OUTPUT play-status).

PROCEDURE sndPlaySoundA EXTERNAL "winmm.dll":
  DEFINE INPUT PARAMETER ic AS CHARACTER.
  DEFINE INPUT PARAMETER ish AS LONG.
  DEFINE RETURN PARAMETER osh AS LONG.
END PROCEDURE.
```

NOTE: You must have a sound driver installed on your machine to play sounds.

The following code sample demonstrates calling the C library function “atoi” to get the value of the character form of an integer. The declaration of the C function being called looks like this: `int atoi(const char *str):`

```
PROCEDURE atoi EXTERNAL "/usr/lib/libc.so.1" :
   DEFINE INPUT PARAMETER b AS MEMPTR.
   DEFINE OUTPUT PARAMETER ret-val AS SHORT.
END PROCEDURE.

DEFINE VARIABLE in-string AS MEMPTR.
DEFINE VARIABLE out-int AS INTEGER INITIAL 0.

SET-SIZE(in-string) = 10.
PUT-STRING(in-string, 1) = "150".

RUN ATOI( INPUT in-string, OUTPUT out-int).

MESSAGE "ATOI RESULT: " out-int VIEW-AS ALERT-BOX.
```

The following procedure defines and displays a shaded ellipse in the current window using DLL functions from the Windows graphics library. This requires initialization of a small structure (ElipRegion).

Note that Progress has no knowledge of any graphics that you create using DLLs. You must ensure that Progress does not refresh the window you are using while the graphics are displayed. Otherwise, the graphics disappear during a window system refresh. (You can help to mitigate this by providing a graphics refresh option within your Progress application.) This procedure displays a preparatory message ("Preparing drawing"), and pauses to realize the current window before calling the DLL routines that display the filled ellipse. The procedure pauses by default before it terminates, allowing the ellipse to remain on the display:

e-dllex3.p *(1 of 3)*

```
/* DLL routine to create an elliptic region */

PROCEDURE CreateEllipticRgnIndirect EXTERNAL "gdi32.dll":
  DEFINE RETURN PARAMETER RegionHandle AS LONG.
  DEFINE INPUT PARAMETER RegionSpec AS MEMPTR.
END PROCEDURE.

/* DLL routine to get drawing object */

PROCEDURE GetStockObject EXTERNAL "gdi32.dll":
  DEFINE RETURN PARAMETER ObjectHandle AS LONG.
  DEFINE INPUT PARAMETER ObjectType AS LONG.
END PROCEDURE.

/* DLL routine to select region into device context */

PROCEDURE SelectObject EXTERNAL "gdi32.dll":
  DEFINE INPUT PARAMETER DeviceHandle AS LONG.
  DEFINE INPUT PARAMETER ObjectHandle AS LONG.
END PROCEDURE.

/* DLL routine to display region */

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

e-dllex3.p *(2 of 3)*

```
/* DLL routine to get handle of window device context */

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

/* DLL routine to release device context handle */

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

/* DLL routine to delete elliptic region */

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

/* Variable Definitions */

DEFINE VARIABLE ElipRegion AS MEMPTR. /* Elliptic region structure */
DEFINE VARIABLE hDevice AS INTEGER.   /* Device context handle   */
DEFINE VARIABLE hObject AS INTEGER.   /* Drawing object handle   */
DEFINE VARIABLE hRegion AS INTEGER.   /* Elliptic region handle  */

DEFINE VARIABLE erLeft AS INTEGER INITIAL 1.    /* Elliptic   */
DEFINE VARIABLE erTop AS INTEGER INITIAL 5.    /* Coordinates */
DEFINE VARIABLE erRight AS INTEGER INITIAL 9.
DEFINE VARIABLE erBottom AS INTEGER INITIAL 13.

/* Allocate and build elliptic region structure */
/* specifying rectangular coordinates of region */

SET-SIZE(ElipRegion) = 4  /* int left   */
                     + 4  /* int top    */
                     + 4  /* int right  */
                     + 4  /* int bottom */
                       .
```

e-dllex3.p *(3 of 3)*

```
PUT-LONG(ElipRegion, erLeft) = 50.
PUT-LONG(ElipRegion, erTop) = 50.
PUT-LONG(ElipRegion, erRight) = 200.
PUT-LONG(ElipRegion, erBottom) = 100.

/* Initialize current window with PAUSE */
/* and display perfunctory message      */

DISPLAY "Preparing drawing..." .
PAUSE.

/* Get device context, region, and drawing object handles */

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

/* Select drawing object and region for device context, */
/* and paint region                                     */

RUN SelectObject(INPUT hDevice, INPUT hObject).
RUN SelectObject(INPUT hDevice, INPUT hRegion).
RUN PaintRgn(INPUT hDevice, INPUT hRegion).

/* Free resources */

SET-SIZE(ElipRegion) = 0.         /* Free region structure  */
RUN ReleaseDC (INPUT CURRENT-WINDOW:HWND, INPUT hDevice).
                                 /* Release device context */
RUN DeleteObject(INPUT hRegion).  /* Delete elliptic region */

/* Wait for user to close window or hit Escape */
WAIT-FOR WINDOW-CLOSE OF CURRENT-WINDOW.
```

6

Windows Dynamic Data Exchange

Dynamic data exchange (DDE) is a general protocol for inter-process communication in Windows. This protocol allows any two Windows applications to communicate as client and server, where the client initiates communications with the server and the server provides data and services to the client.

Progress supports DDE communications acting as a DDE client only. Using 4GL DDE statements, your Progress application can initiate and maintain communications with any other Windows application with DDE server capability. This communications includes exchanging data with the server and directing the server to execute server-defined commands. Your Progress application can communicate with different server applications during a session, and can even access entirely new server applications without modifying your existing Progress code.

The following sections describe how to use DDE in your Progress applications. For more information on DDE concepts and facilities, see the Windows programming documentation on DDE.

This chapter contains the following sections:

6.1 Using the Dynamic Data Exchange Protocol

The DDE protocol establishes communications between two applications through a *conversation*. A conversation provides a unique conduit through which they can exchange information. One application is the *DDE server* and the other the *DDE client*. It is the responsibility of the DDE client to open and manage conversations with a DDE server. The DDE server responds to client requests to send or receive information, and to execute server commands on the client's behalf.

A Progress application is always the DDE client and can only open conversations with applications that act as DDE servers. Thus, two Progress clients cannot use DDE to converse with each other (except through an intermediary server).

6.1.1 The Course Of a DDE Conversation

All DDE conversations follow a pattern of execution that varies somewhat with the application. In general, to open and manage a DDE conversation, your application must complete the following tasks.

1. Select a named frame to use as a conversation endpoint (*DDE frame*). DDE attributes of this frame maintain the status of each conversation you open with it.

   NOTE: A DDE frame cannot be a dialog box (no VIEW-AS DIALOG-BOX option).

2. Ensure that the DDE server application is running in your Windows environment and that the DDE frame for your application is realized.

3. Open the conversation to the server application, specifying the DDE frame, application, and topic names. A topic is a category defined by the server that includes specific data items or commands that the client can access.

4. If you want to, send commands to the DDE server that define additional topics or otherwise prepare the server to open DDE conversations. (You typically open an initial conversation for the System topic to execute these commands.)

5. Send and receive data values between your Progress client and DDE server using the data items associated with your topic of conversation. You can converse with server data items on demand, or set up links that let your Progress application automatically receive the data when server data items change value.

6. To close a conversation, remove any links established to associated data items and terminate the conversation.

6.1.2 4GL Statements For DDE Conversations

Progress provides several 4GL statements to open and manage DDE conversations. Table 6–1 lists these DDE statements.

Table 6–1: DDE Statements

Statement	Description
DDE ADVISE	Creates or removes a link to a DDE server data item. Creating a link allows Progress to trigger a DDE-NOTIFY event when the data item changes value.
DDE EXECUTE	Sends one or more application commands to the DDE server to execute.
DDE GET	Retrieves the new value of a linked data item in response to a DDE-NOTIFY event for the data item. You typically invoke this command in a DDE frame trigger block for the DDE-NOTIFY event.
DDE INITIATE	Opens a DDE conversation.
DDE REQUEST	Requests the current value of a server data item.
DDE SEND	Sends a new value to a server data item.
DDE TERMINATE	Closes a DDE conversation.

The following sections provide more information on how Progress structures DDE conversations and how to use these 4GL statements to open, manage, and close DDE conversations. For a complete description of each statement, see the *Progress Language Reference*.

6.2 Structuring a DDE Conversation In Progress

In a DDE conversation, the Progress client sends and receives data values between Progress fields, variables, and expressions at one end and corresponding data items in the server application at the other end. The client can also send commands to the server using whatever language or format that the server recognizes. The documentation for the server application generally tells you how to set up DDE conversations. This information includes how to address the server from any client, such as Progress. In general, Progress structures all DDE conversations within a standard hierarchy.

6.2.1 Conversation Hierarchy

Any DDE conversation is structured in a hierarchy that consists of an *application name* that identifies the DDE server, a *topic name* within the application, and one or more *data item names* within the topic. The application and topic name together with a DDE frame uniquely identifies the conversation, which Progress also assigns a unique *channel number*.

The application name is a name defined by the server that is unique to other DDE server applications. This is usually the filename of the executable without the extension (for example, EXCEL for Microsoft Excel). The topic name identifies a category used to group server data items. If the data items are in a file accessed by the server, the topic name might be the filename (for example, a worksheet name such as Sheet1 in Excel). The data item names identify the data items that the server defines for the topic. These names generally follow a convention defined by and used in the server application itself (for example, the row and column coordinates of a worksheet cell, such as R3C12 in Excel). Once the Progress client opens a conversation, it might exchange values with any server data item associated with the topic of conversation.

Figure 6–1 shows the conversational relationship between a Progress client and two DDE servers, Excel and a Visual Basic application. In this example, there are three open conversations—two with Excel and one with the Visual Basic application.

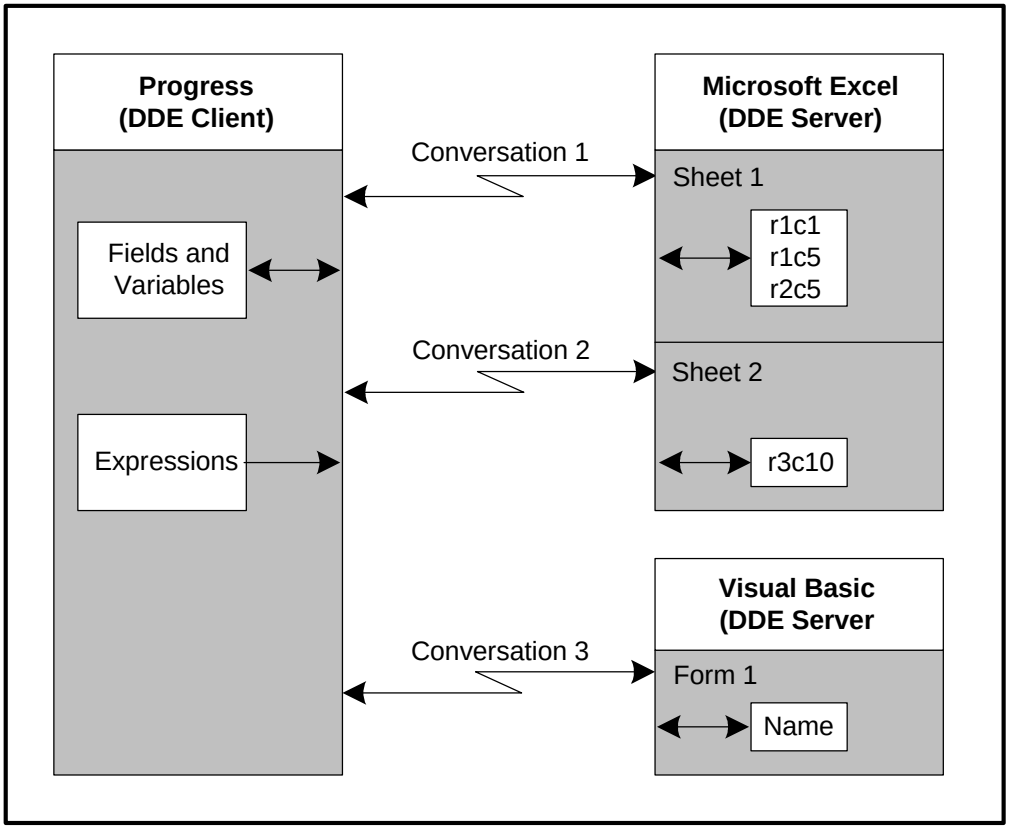


Figure 6–1: Progress DDE Conversations

During a conversation, Progress can send and receive data between a Progress field or variable and a server data item, or send the value of a Progress expression to a server data item. Typically, you cannot access a server expression from Progress unless the expression is directly associated with a data item—for example, a worksheet expression that defines the value of a cell in Excel. In this case, while Progress can receive the value of the cell expression from the data item (cell), it cannot send a new value to that cell, since its value is determined by the server expression. However, DDE server applications, such as Excel, usually allow clients to completely redefine server topics and data items by executing server commands using the DDE. (The `e-ddeex1.p` procedure at the end of this chapter demonstrates this capability.)

6.3 Defining Conversation Endpoints—DDE Frames

Progress uses named frames as client endpoints for DDE conversations. A DDE frame, in effect, owns the client end of a conversation. You specify the DDE frame for a conversation when you open it with the DDE INITIATE statement.

A DDE frame can be visible or hidden during application execution. A hidden DDE frame effectively insulates the user from the conversation, preventing any unintended user intervention. However, the frame must be realized before you can open a conversation with it. That is, you must make it visible before hiding it (set the frame VISIBLE attribute = TRUE; then the HIDDEN attribute = TRUE).

A DDE frame can be a client endpoint for more than one DDE conversation, and you can use multiple DDE frames to open multiple conversations with the same application and topic. Your decision to use one or more DDE frames depends entirely on how you decide to allocate frames for your application and distribute DDE conversations among them. Otherwise, it makes no difference whether you use one or many DDE frames for conversation endpoints.

Each DDE frame provides a set of frame attributes that maintain the status of conversations owned by the frame. Table 6–2 lists these DDE frame attributes.

Table 6–2: DDE Frame Attributes

Attribute	Type	Readable	Setable
DDE-ERROR	INTEGER	√	–
DDE-ID	INTEGER	√	–
DDE-ITEM	CHARACTER	√	–
DDE-NAME	CHARACTER	√	–
DDE-TOPIC	CHARACTER	√	–

Each time a conversational exchange occurs—that is, each time a DDE statement executes or Progress posts a DDE-NOTIFY event—Progress updates the DDE attributes of the frame that owns the conversation. These attributes are especially useful for determining the exact nature of a DDE run-time error (DDE-ERROR) and identifying the data item that triggered a DDE-NOTIFY event (DDE-NAME, DDE-TOPIC, and DDE-ITEM). (For more information on DDE-NOTIFY events, see the “Exchanging Data In Conversations” section.)

6.3.1 DDE-ERROR

This attribute records the error condition returned by the last exchange for the frame. Table 6–3 lists the possible errors returned by DDE conversation exchanges.

Table 6–3: Progress DDE Errors

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

6.3.2 Other DDE Attributes

DDE-ID

This attribute records the channel number of the conversation that had the most recent exchange.

DDE-ITEM

This attribute records the name of the server data item affected by the most recent conversation exchange.

DDE-NAME

This attribute records the application name of the conversation that had the most recent conversation exchange.

DDE-TOPIC

This attribute records the topic name of the conversation that had the most recent conversation exchange.

6.4 Opening DDE Conversations

To open a DDE conversation in a Progress client, the application and user must follow these steps.

1. Execute and prepare the server application to accept DDE conversations. This includes making all necessary server topics available to the client. This can be done either from inside the client or externally in Windows.
2. Define one or more named frames to use as DDE frames.
3. Open each conversation using the DDE INITIATE statement.

6.4.1 Preparing the Server Application

Before opening a DDE conversation from the client, you must ensure that the server application is open on the Windows desktop, and that any server preparations required to make application topics available to the Progress client are complete. The Progress client can open the server application by either by invoking the 4GL OS-COMMAND NO-WAIT statement or by executing the `WinExec()` program load function from the Window's kernel dynamic link library (DLL), `kernel32.dll`. The program load function provides additional features that include specifying the startup window state and returning a value that indicates whether the application actually started. (For more information on accessing Windows DLLs from Progress, see Chapter 5, "Shared Library and DLL Support.")

For example, the following code fragment defines the interface to `WinExec()`, and uses it to load Microsoft Notepad:

e-ddeex2.p *(1 of 2)*

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

PROCEDURE WinExec EXTERNAL "KERNEL32.DLL":
  DEFINE INPUT  PARAMETER ProgramName AS CHARACTER.
  DEFINE INPUT  PARAMETER VisualStyle AS LONG.
  DEFINE RETURN PARAMETER StatusCode  AS LONG.
END PROCEDURE.

/*************************************************/
/* NOTE: VisualStyle parameters are as follows:   */
/*       1 = Normal         2 = Minimized         */
/*************************************************/

RUN WinExec (INPUT "NOTEPAD", INPUT 1, OUTPUT ReturnCode).
IF ReturnCode >= 32 THEN
  MESSAGE "Application was Started" VIEW-AS ALERT-BOX.
ELSE
  MESSAGE "Application Failed:" ReturnCode VIEW-AS ALERT-BOX.

/*************************************************************************/
/* RETURN CODE DESCRIPTION                                               */
/* If the function is successful, the return value from WinExec          */
/* identifies the instance of the loaded module. Otherwise, the          */
/* return value is an error value between 0 and 32.                      */
/*                                                                       */
/*  0   System was out of memory, executable file was corrupt, or        */
/*        relocations were invalid.                                      */
/*  2    File was not found.                                             */
/*  3    Path was not found.                                             */
/*  5   Attempt was made to dynamically link to a task, or there         */
/*         was a sharing or network protection error.                    */
/*  6   Library required separate data segments for each task.           */
/*  8   There was insufficient memory to start the application.          */
```

e-ddeex2.p *(2 of 2)*

```
/* 10    Windows version was incorrect.                                    */
/* 11   Executable file was invalid. Either it was not a Windows      */
/*        application or there was an error in the .EXE image.        */
/* 12   Application was designed for a different operating system.    */
/* 13    Application was designed for MS-DOS 4.0.                     */
/* 14    Type of executable file was unknown.                         */
/* 15    Attempt was made to load a real-mode application             */
/*         (developed for an earlier version of Windows).             */
/* 16   Attempt was made to load a second instance of an executable   */
/*       file containing multiple data segments that were not         */
/*        marked read-only.                                           */
/* 19   Attempt was made to load a compressed executable file.        */
/*        The file must be decompressed before it can be loaded.      */
/* 20   Dynamic link library (DLL) file was invalid.  One of the DLLs */
/*        required to run this application was corrupt.               */
/* 21   Application requies Microsoft Windows 32-Bit extensions.      */
/*********************************************************************/
```

Depending on the application, the client might then open an initial conversation for the System topic and execute server commands to initialize additional topics.

6.4.2 Defining DDE Frames

To open a conversation, you must first define a named frame to use as the DDE frame for the conversation. Make sure that you define the DDE frame in a scope large enough to complete the intended conversation. If you are unsure, use a FORM or DEFINE FRAME statement to define the frame for the scope of the procedure(s) that invoke DDE exchanges. For example, the following code defines and enables a frame for the procedure scope to start and manage conversations with Microsoft Excel:

```
DEFINE VARIABLE listx AS CHARACTER VIEW-AS SELECTION-LIST SIZE 36 BY 5.
DEFINE VARIABLE ed    AS CHARACTER VIEW-AS EDITOR SIZE 20 by 2.

DEFINE BUTTON bq LABEL "Quit".
DEFINE BUTTON bg LABEL "Start Excel".
                    .
                    .
                    .
DEFINE FRAME MainFrame
  SKIP(1) SPACE(1) bq SPACE(1) bg SPACE(1) SKIP(1)
  SPACE(1) listx LABEL "DDE History" SPACE(1) SKIP(1)
SPACE(1) ed LABEL "Cell R4C2 (Row 4 Col B)" SKIP(1)
WITH SIDE-LABELS.
ENABLE ALL WITH FRAME MainFrame TITLE "Main Frame".
                    ...
```

If you want your application to completely hide DDE conversations from the user, always define your DDE frames as procedure-scoped static frames without fields and set their HIDDEN attributes to TRUE after realizing the frames. This prevents the user from doing anything that might compromise DDE communications, such as invoking an option that inadvertently destroys the frame in mid-conversation.

6.4.3 Initiating a Conversation

After defining the DDE frame, invoke the DDE INITIATE statement for the frame, server name, and an available server topic. Most server applications provide a System topic that is available at startup. The data items for this topic generally include application command strings and other information to support DDE interactions with the application. In Progress, you can use this System topic to make other topics available to the client using the DDE EXECUTE statement. After you have made these topics available, you can open additional conversations for them.

The DDE INITIATE statement retrieves a channel number that uniquely identifies the conversation. You use this channel number with other DDE statements to invoke all other exchanges in the conversation.

For example, the following code fragment opens a conversation with Microsoft Excel's System topic, returning the channel number to sys. It uses the sys conversation to invoke the Excel NEW command, creating an initial Excel worksheet (Sheet1). It then opens a conversation with the Sheet1 topic, returning the channel number to sheet:

```
DEFINE VARIABLE sys AS INTEGER.
DEFINE VARIABLE sheet AS INTEGER.
                  .
                  .
                  .
DDE INITIATE  sys FRAME FRAME MainFrame:HANDLE
            APPLICATION "Excel" TOPIC "System".
DDE EXECUTE    sys COMMAND "[new(1)]".
DDE INITIATE  sheet FRAME FRAME MainFrame:HANDLE
            APPLICATION "Excel" TOPIC "Sheet1".
                  .
                  .
                  .
```

6.5 Exchanging Data In Conversations

Once you have opened a DDE conversation, you can converse on the topic using four types of exchanges. These exchange types include:

- **Execute** — The DDE client executes server commands using the DDE EXECUTE statement.
- **Request** — The DDE client retrieves data from the server using the DDE REQUEST statement.
- **Send** — The DDE client sends data to the server using the DDE SEND statement.
- **Advise** — The DDE client creates *advise links* to server data items using the DDE ADVISE statement. Advise links direct the DDE server to monitor the data item. When any linked data item changes value, your Progress application is notified so it can retrieve the value using the DDE GET statement.

Execute, request, and send exchanges are each *demand-driven*. When you invoke the exchange, Progress waits for the exchange to complete. A demand-driven exchange works like a FIND statement, which waits until Progress retrieves the desired record from the database. Advise exchanges are *event-driven*. Once an advise link is created for a data item, the Progress client continues execution and retrieves the data item value "automatically" as it changes. You can invoke both demand, and event-driven exchanges in the same conversation and for the same data item. The following sections describe how to implement these exchanges.

6.5.1 Demand-driven Exchanges

In a demand-driven exchange, the Progress client sends data to or requests data from the server, and waits for each exchange to complete before continuing. The DDE EXECUTE, DDE REQUEST, and DDE SEND statements all invoke demand-driven exchanges.

Execute Exchanges

You can use the DDE EXECUTE statement to send commands for the DDE server to execute, as shown previously in the "Initiating a Conversation" section. Typically, you send server commands in a conversation opened for the System topic of the server application. However, the server might support commands for any topic that it provides. For example, Excel lets you select a range of cells (data items) in a worksheet topic. In effect, server commands are data items that it executes when sent with the DDE EXECUTE statement. For a more complete example of this type of exchange, see the "Closing DDE Conversations" section.

The System topic for your server application might also provide other data items that you can read or set using request and send exchanges. These data items typically provide server options or status information to the client.

Request and Send Exchanges

You can use the DDE REQUEST statement to retrieve any server data item as a character string value. If you plan to convert the string value to another Progress data type (for example, DATE), you must convert the string to the correct format.

You can use the DDE SEND statement to send a character string value to any server data item. You must ensure that the string conforms to a format acceptable to the server data item.

You might use request and send exchanges to create new database records from the rows of a worksheet, or iteratively read a Progress database and fill out worksheet rows from selected records. For example, the following code fragment fills out an Excel worksheet using three fields from the customer table of the sports database:

```
DEFINE VARIABLE rowi AS INTEGER.
DEFINE VARIABLE sheet AS INTEGER.
DEFINE VARIABLE itemn AS CHARACTER.
              .
              .
              .
DDE SEND sheet SOURCE "Name" ITEM "r1c1".
DDE SEND sheet SOURCE "Balance" ITEM "r1c2".
DDE SEND sheet SOURCE "State" ITEM "r1c3".
rowi = 2.
FOR EACH customer WHERE balance > 10000 BY balance:
  itemn = "R" + STRING(rowi) + "C1".
  DDE SEND sheet SOURCE customer.name ITEM itemn.
  itemn = "R" + STRING(rowi) + "C2".
  DDE SEND sheet SOURCE STRING(customer.balance) ITEM itemn.
  itemn = "R" + STRING(rowi) + "C3".
  DDE SEND sheet SOURCE STRING(customer.state) ITEM itemn.

  rowi = rowi + 1.
END.
```

In the example, the first three DDE statements send column titles to the first three columns of the first row of the worksheet. Then, for each customer record showing more than $10,000 in payables, the DDE statements send the customer’s name, balance, and state of residence to the appropriate columns of the next row in the worksheet.

6.5.2 Event-driven Exchanges

In an event-driven exchange, the DDE server application sends the value of the data item to the client whenever the value of the data item changes. Each server data item you set up for event-driven exchange actually participates in a series of exchanges, totally dependent on how often the data item changes. If the data item never changes value, no exchange occurs.

Steps To Setting Up Event-driven Exchanges—Advise Links

To set up and manage event-driven exchanges for a data item, your application must complete these steps:

1. Specify the server data item in a DDE ADVISE statement, using the START option. This creates an advise link to the data item from Progress. From this point in the procedure, the DDE server monitors the specified data item, notifying Progress whenever its value changes.

2. Add an ON DDE-NOTIFY OF FRAME *DDEframe* statement, where *DDEframe* is the name of the frame that owns the conversation. In the trigger block for the statement, invoke the DDE GET statement to retrieve the new value of the data item (just like a DDE REQUEST) and process it as you want. Progress triggers this DDE-NOTIFY event and posts it to the appropriate DDE frame when notified of the value change. The Progress client application then executes the event trigger when it blocks for I/O or invokes the PROCESS EVENTS statement. If you have more than one advise link established for the conversation, you can determine what data item changed by checking the value of the DDE-ITEM attribute of the DDE frame.

   NOTE: In general, do not block for I/O (for example, invoke an UPDATE statement) or invoke a PROCESS EVENTS statement within the trigger for a DDE-NOTIFY event. This can cause Progress to update the DDE frame attributes for a new DDE-NOTIFY event before you have completed the processing for a prior event.

3. At any point in the procedure, if you want to stop event-driven exchanges for the data item, specify the data item in a DDE ADVISE statement using the STOP option. This removes the advise link and directs the server to cease monitoring value changes in the data item. This does not terminate the conversation in any way and you can continue to access the data item with other exchanges or create another advise link to the data item.

Coordinating DDE Client/Server Communications

If you use event-driven exchanges, you might also want to set a value for the MULTITASKING-INTERVAL attribute of the SESSION system handle. The value of this attribute determines how often Progress checks for application events. An appropriate value can help your client application interact more smoothly with its DDE server. For more information, see the *Progress Language Reference*.

Applications For Event-Driven Exchanges

You might use event-driven exchanges to tie data item values in a Progress application to those in a spreadsheet, calender scheduler, or some other database application that might be running in your application environment. For example, the following code fragment retrieves the latest value of the cell at row 4, column 2 (the "B" column) in an Excel worksheet and stores it in a Progress editor field:

```
DEFINE VARIABLE sheet AS INTEGER.
DEFINE VARIABLE sty AS CHARACTER.
DEFINE VARIABLE ed AS CHARACTER
  VIEW-AS EDITOR SIZE 20 by 2.

ON DDE-NOTIFY OF FRAME MainFrame
DO:
  DDE GET sheet TARGET sty ITEM "r4c2".
  sty = SUBSTR(sty, 1, 20).           /* Drop the CR/LF */
  ed:VALUE = IN FRAME MainFrame = sty.
END.
               .
               .
               .
DDE ADVISE sheet START ITEM "r4c2".
               .
               .
               .
```

Multiple Conversations With Advise Links

If a DDE frame owns more than one conversation with advise links, you can get the channel number of the conversation and the name of the data item to which a DDE-NOTIFY event applies from the DDE-ID and DDE-ITEM attributes of the frame. This is enough information to retrieve the value that triggered the event with the DDE GET statement. However, you might also need the application and topic names to fully identify the item from which you are retrieving data. This is necessary if the different conversations (different topics and/or applications) include links to data items with the same name. You can obtain the application and topic names related to the current DDE-NOTIFY event from the DDE-NAME and DDE-TOPIC attributes of the frame. For more information on DDE frame attributes, see the "Defining Conversation Endpoints—DDE Frames" section.

However, you might find it simpler to manage event-driven exchanges by using a separate DDE frame for each advise link. In this case, you initiate an additional conversation for each advise link that you want to establish for the same application and topic, using a separate DDE frame for each conversation. You then set up each advise link using its own conversation ID. When a DDE-NOTIFY event occurs for a linked data item, you do not have to check for the source of the event, because each data item has its own frame trigger. The following code fragment shows the essential elements of this technique to link two data cells in the same Excel worksheet to two Progress variables (`quote1` and `quote2`):

```
DEFINE VARIABLE link1 AS INTEGER.
DEFINE VARIABLE link2 AS INTEGER.
DEFINE VARIABLE quote1 AS CHARACTER.
DEFINE VARIABLE quote2 AS CHARACTER.
DEFINE FRAME Flink1.
DEFINE FRAME Flink2.

ON DDE-NOTIFY OF FRAME Flink1 DO:
    DDE GET link1 TARGET quote1 ITEM "R2C1".
                .
                .
                .
ON DDE-NOTIFY OF FRAME Flink2 DO:
    DDE GET link2 TARGET quote2 ITEM "R2C2".
                .
                .
                .
DDE INITIATE link1 FRAME Flink1:HANDLE
    APPLICATION "EXCEL" TOPIC "Sheet1".
DDE INITIATE link2 FRAME Flink2:HANDLE
    APPLICATION "EXCEL" TOPIC "Sheet1".

DDE ADVISE link1 START ITEM "R2C1".
DDE ADVISE link2 START ITEM "R2C2".
```

6.6 Closing DDE Conversations

You can close a DDE conversation in one of three ways:

- Invoke the DDE TERMINATE statement for the conversation.
- Leave the scope of the DDE frame that owns the conversation.
- Terminate or remove the DDE server or server topic associated with the conversation.

Regardless of how you close a DDE conversation, once closed, the channel number for that conversation is no longer available for further exchanges. Terminating the conversation with the DDE TERMINATE statement has no effect on other conversations open for the same DDE server or frame. They continue without interruption.

In general, if your Progress client creates and manages a server environment for the conversation, it should also clean up that environment when closing the conversation. For example, the following code fragment cleans up a Microsoft Excel server environment when the user presses a **QUIT** button. The client removes an advise link to the worksheet and closes the conversation for that worksheet topic (sheet). Then using the System conversation (sys), it executes server commands that close the documents opened for the conversation, clear Excel error checking, and instruct the server to shut itself down:

```
DEFINE VARIABLE sys AS INTEGER.
DEFINE VARIABLE sheet AS INTEGER.

DEFINE BUTTON bq LABEL "Quit".
                     .
                     .
                     .
ON CHOOSE OF bq IN FRAME MainFrame
DO:
                     .
                     .
                     .
  DDE ADVISE sheet STOP ITEM "r4c2".
  DDE TERMINATE sheet.
  DDE EXECUTE sys COMMAND "[activate(~"sheet1~")]".
  DDE EXECUTE sys COMMAND "[close(0)]" 0).
  DDE EXECUTE sys COMMAND "[activate(~"chart1~")]".
  DDE EXECUTE sys COMMAND "[close(0)]".
  DDE EXECUTE sys COMMAND "[error(0)]".
  DDE EXECUTE sys COMMAND "[quit()]".
END.
                     .
                     .
                     .
```

Note the tilde (~) used to escape embedded quotes in strings—for example, `~"sheet1~"`.

6.7 DDE Example

The following Progress procedure uses the DDE facility to build a Microsoft Excel worksheet of customer balances from the sports database and display the customer payables distribution in an Excel pie chart. It incorporates most of the code examples in the previous sections.

The visible DDE frame displays a selection list (DDE History) showing the server, topic, and item name for each exchange as it occurs. (Note the custom selection list delimiter (|) used in place of the default comma delimiter that appears in commands.) When you manually change the balance value in cell `r4c2` of the worksheet, Progress uses an advise link to retrieve and display the new value in the field labeled `Cell R4C2 (Row 4 Col B):`

NOTE: For the following example to work, you must have Excel in your path or specify the full Excel pathname for the prog_name parameter when you run the WinExec procedure to start Excel. This example assumes that you have Excel in the default Microsoft Office directory.

e-ddeex1.p *(1 of 3)*

```
DEFINE VARIABLE listx AS CHARACTER VIEW-AS SELECTION-LIST SIZE 36 BY 5.
DEFINE VARIABLE rowi  AS INTEGER.
DEFINE VARIABLE sys   AS INTEGER.
DEFINE VARIABLE sheet AS INTEGER.
DEFINE VARIABLE itemn AS CHARACTER.
DEFINE VARIABLE sty   AS CHARACTER.
DEFINE VARIABLE ed    AS CHARACTER VIEW-AS EDITOR SIZE 20 by 2.
DEFINE VARIABLE log_i  AS LOGICAL.
DEFINE VARIABLE excelon AS LOGICAL INITIAL FALSE.

DEFINE BUTTON bq LABEL "Quit".
DEFINE BUTTON bg LABEL "Start Excel".

DEFINE FRAME MainFrame
    SKIP(1) SPACE(1) bq SPACE(1) bg SPACE(1) SKIP(1)
    SPACE(1) listx LABEL "DDE History" SPACE(1) SKIP(1)
    SPACE(1) ed LABEL "Cell R4C2 (Row 4 Col B)" SKIP(1)
WITH SIDE-LABELS.
ENABLE ALL WITH FRAME MainFrame TITLE "Worksheet Monitor".
listx:DELIMITER = "|". /* No server commands use "|" */
```

e-ddeex1.p *(2 of 3)*

```
PROCEDURE WinExec EXTERNAL "kernel32.dll": /* Run Windows application */
    DEFINE INPUT PARAMETER prog_name AS CHARACTER.
    DEFINE INPUT PARAMETER prog_style AS LONG.
END PROCEDURE.

PROCEDURE LogExchange: /* Log latest DDE in selection list */
    log_i = listx:ADD-LAST(FRAME MainFrame:DDE-NAME + " " +
                           FRAME MainFrame:DDE-TOPIC + " " +
                           FRAME MainFrame:DDE-ITEM) IN FRAME MainFrame.
END PROCEDURE.

ON CHOOSE OF bq IN FRAME MainFrame
DO:
   IF (excelon = FALSE) THEN RETURN.

   DDE ADVISE sheet STOP ITEM "r4c2". RUN LogExchange.
   DDE TERMINATE sheet. RUN LogExchange.
   DDE EXECUTE sys COMMAND "[activate(~"sheet1~")]". RUN LogExchange.
   DDE EXECUTE sys COMMAND "[close(0)]". RUN LogExchange.
   DDE EXECUTE sys COMMAND "[activate(~"chart1~")]". RUN LogExchange.
   DDE EXECUTE sys COMMAND "[close(0)]". RUN LogExchange.
   DDE EXECUTE sys COMMAND "[error(0)]". RUN LogExchange.
   DDE EXECUTE sys COMMAND "[quit()]". RUN LogExchange.
END.

ON CHOOSE OF bg IN FRAME MainFrame
DO:
   /* INPUT: 1=normal 2=minimized */
   RUN WinExec (INPUT "C:\Program Files\Microsoft Office\Office\Excel /e",
INPUT 2).
   excelon = TRUE.

   DDE INITIATE sys FRAME FRAME MainFrame:HANDLE
      APPLICATION "Excel" TOPIC "System". RUN LogExchange.
   IF sys = 0 THEN
   DO:
      MESSAGE "Excel not available".
      RETURN.
   END.
```

e-ddeex1.p *(3 of 3)*

```
   DDE EXECUTE sys COMMAND "[new(1)]". RUN LogExchange.
   DDE INITIATE sheet FRAME FRAME MainFrame:HANDLE
      APPLICATION "Excel" TOPIC "Sheet1". RUN LogExchange.

   DDE SEND sheet SOURCE "Name" ITEM "r1c1". RUN LogExchange.
   DDE SEND sheet SOURCE "Balance" ITEM "r1c2". RUN LogExchange.
   DDE SEND sheet SOURCE "State" ITEM "r1c3". RUN LogExchange.
   rowi = 2.

   FOR EACH customer WHERE balance < 5000 BY balance:
      itemn = "R" + STRING(rowi) + "C1".
      DDE SEND sheet SOURCE customer.name ITEM itemn.
         RUN LogExchange.
      itemn = "R" + STRING(rowi) + "C2".
      DDE SEND sheet SOURCE STRING(customer.balance) ITEM itemn.
         RUN LogExchange.
      itemn = "R" + STRING(rowi) + "C3".
      DDE SEND sheet SOURCE STRING(customer.state) ITEM itemn.
         RUN LogExchange.
      rowi = rowi + 1.
   END.

   DDE EXECUTE sheet COMMAND "[select(~"C1:C2~")]". RUN LogExchange.
   DDE EXECUTE sys COMMAND "[column.width(,,,3)]". RUN LogExchange.
   DDE EXECUTE sys COMMAND "[format.number(~"$#,##0~")]". RUN LogExchange.
   DDE EXECUTE sheet COMMAND "[select(~"C1:C2~")]". RUN LogExchange.
   DDE EXECUTE sys COMMAND "[new(2,1,1)]". RUN LogExchange.
   DDE EXECUTE sys COMMAND "[activate(~"chart1~")]". RUN LogExchange.
   DDE EXECUTE sys COMMAND "[gallery.3d.pie(3)]". RUN LogExchange.
   DDE EXECUTE sys COMMAND "[app.restore()]". RUN LogExchange.
   DDE EXECUTE sys COMMAND "[arrange.all()]". RUN LogExchange.

   DDE ADVISE sheet START ITEM "r4c2". RUN LogExchange.
END.

ON DDE-NOTIFY OF FRAME MainFrame
DO:
   DDE GET sheet TARGET sty ITEM "r4c2". RUN LogExchange.
   sty = SUBSTR(sty, 1, 20).             /* Drop the CR/LF */
   ed:SCREEN-VALUE IN FRAME MainFrame = sty.
END.

WAIT-FOR CHOOSE OF bq IN FRAME MainFrame.
```

7

Using COM Objects In the 4GL

Both ActiveX Automation objects and ActiveX controls are COM objects—objects that conform to the specifications of the Microsoft Component Object Model (COM). As such, COM objects have common features that govern the programming of both Automation objects and ActiveX controls in the 4GL.

This chapter describes the common features of COM objects supported by Progress and how to work with them in a 4GL application and your development environment, in the following sections:

For an overview of the architectural elements that support COM objects in Progress, see Chapter 1, “Introduction.” For information on COM object support that is unique for Automation objects, see Chapter 8, “ActiveX Automation Support” and for ActiveX controls, see Chapter 9, “ActiveX Control Support.” The information in this chapter applies equally to Automation objects and ActiveX controls, except where noted.

7.1 How COM Objects Differ From Progress Widgets

COM objects and Progress widgets are two different types of objects. Although they can provide similar capabilities for an application, their function and management in the 4GL is quite different.

7.1.1 Functionality

COM objects are provided by third-party vendors as well as by Progress Software Corporation. They have standard features that allow them to be accessed from many different programming environments. The capabilities of COM objects vary widely. Many provide some user interface component, while others provide such functionality as HTTP support and internet access. However, because COM objects are independent of Progress, they have no direct integration with Progress's data management facilities, such as database fields, formats, and validation.

Progress widgets are objects that are completely defined by Progress and are not directly accessible by any other programming environment. Their capabilities focus primarily on user interface function. Unlike COM objects, Progress widgets are fully integrated into the programming and data management facilities of the 4GL. This means that a variety of default Progress behaviors apply to widgets that do not apply to COM objects.

7.1.2 4GL Mechanisms

Ultimately, the differences between COM objects and widgets require separate but related mechanisms in the 4GL to work with them. The most fundamental of these mechanisms are the handles you use to access COM objects and widgets. You must access widgets with widget handles and COM objects with component handles. These handles have different data types that represent the different capabilities that they support—the WIDGET-HANDLE data type for widget handles and the COMPONENT-HANDLE (or COM-HANDLE) data type for COM objects.

COM-HANDLE values have a data type that is compatible with most other Progress data types. For component handles, Progress does the data conversion automatically, unlike widget handles which require the use of Progress data conversion functions, like STRING or INTEGER.

Table 7–1 compares each type of object (COM object and Progress widget) and summarizes how it is supported in the 4GL.

Table 7–1: COM Objects and Progress Widgets Compared

Feature	COM Object Support	Progress Widget Support
Compilation	No compile-time checking	Compile-time checking for attribute and method references
Instantiation	Dynamic, with different mechanisms for Automation objects and ActiveX controls	Static or Dynamic, depending on the widget and application
Handle type	COM-HANDLE	WIDGET-HANDLE
Handle validity	VALID-HANDLE function	VALID-HANDLE function
Access to data	Properties (optionally indexed)	Attributes
Access to behavior	Methods (with unnamed parameters)	Methods (with unnamed parameters)
Access to events	Event procedures (with unnamed parameters)	Triggers
Integration with 4GL data	Extensive automatic mapping between COM data types and Progress data types	Full integration with Progress data types and 4GL data management
Color management	RGB color values (RGB-VALUE function, GET-RGB-VALUE method on the COLOR-TABLE system handle)	Index numbers of RGB color values in the Progress color table (COLOR-TABLE system handle)
Dynamic object management	RELEASE OBJECT statement	DELETE Widget statement

For more information on Progress widgets and the 4GL mechanisms for managing them, see the information on widgets and handles in the *Progress Programming Handbook*. The rest of this chapter describes the equivalent mechanisms that support COM objects.

7.2 Obtaining Access To COM Objects

To access a COM object, you have to define a component handle for it and set the handle value using the appropriate object instantiation:

```
DEFINE VARIABLE hCOMobject AS COM-HANDLE.
/* ... Instantiate COM object, setting hCOMobject ... */
/* ... Access COM object properties and methods using hCOMobject ... */
/* ... Access ActiveX control events using OCX event procedures ... */
```

To instantiate an Automation object, you use the CREATE Automation Object statement. For more information, on instantiating Automation objects, see Chapter 8, "ActiveX Automation Support."

To instantiate an ActiveX control, you use the AppBuilder at design time to select and configure the control, and to generate the 4GL that instantiates the control at run time. This AppBuilder-generated 4GL includes the CREATE Widget statement to create the control-frame widget and the LoadControls( ) method to associate the control instance with the control-frame COM object. For more information on instantiating ActiveX controls, see Chapter 9, "ActiveX Control Support."

Access to COM object properties and methods is the same for both Automation objects and ActiveX controls. Progress supports event management for both Automation objects and ActiveX controls. For information on ActiveX Automation object event management, see Chapter 8, "ActiveX Automation Support." For information on ActiveX control event management, see Chapter 9, "ActiveX Control Support."

7.3 Accessing COM Object Properties and Methods

Progress supports direct access to COM object properties and methods. This support extends the syntax used for widget attribute and method references. Other than syntax validation, Progress does no compile-time checking of property and method references. All other property and method validation occurs at run time, when all COM objects are (dynamically) instantiated. Thus at run time, Progress dispatches each property or method reference to the COM object for evaluation and execution.

7.3.1 Property and Method Syntax

The syntax to access properties and methods is similar to the syntax to access widget attributes and methods. However, where widget references use widget handles, COM object references use component handles. Component handles support an extended syntax that allows you to:

- Chain component handle references to properties and methods
- Specify indexes on properties
- Specify mappings between Progress data types and COM data types for method output parameters and property settings
- Specify additional options for method parameters and return values

The following syntax diagrams describe the syntax for method and property references. These diagrams are equivalent to the syntax presented in the *Progress Language Reference*, but describe method and property references in a more top-down fashion. (See the information on attributes and methods in the *Progress Language Reference*.)

NOTE: All COM object errors are translated to Progress errors. To suppress any error messages generated by a COM object method or property reference, you can specify the NO-ERROR option in any statement that includes the method or property reference.

Method Reference

This is the syntax for a COM object method reference:

SYNTAX

```
[ NO-RETURN-VALUE ]
    COMhdl-expression:Method-Name-Reference
```

NO-RETURN-VALUE

Required for some methods that do not have a return value.

NO-RETURN-VALUE prevents Progress from expecting a possible return value. This option is appropriate only if the method does not have a return value. Whether a method call requires this option depends on the COM object. Some COM objects require that the caller knows there is no return value. In this case, you must specify the option. Many more robust COM objects do not require this option.

If the method requires NO-RETURN-VALUE and you don't specify it, the COM object generally returns an error. For example, Word for Windows 95 Version 7.0 returns an error similar to "Non-function called as function."

NOTE: If the method has a return value, you must not invoke it in a statement with the NO-RETURN-VALUE option. In this case, ignore the return value without specifying this option, as in the second call to the SetDate( ) method.

COMhdl-expression

An expression that returns a component handle to the COM object that owns the method specified by *Method-Name-Reference*.

Method-Name-Reference

Specifies a single COM object method that might return a value.

Property Reference

This is the syntax for a COM object property reference:

SYNTAX

```
COMhdl-expression:Property-Name-Reference
```

COMhdl-expression

An expression that returns a component handle to the COM object that owns the property specified by *Property-Name-Reference*.

Property-Name-Reference

Specifies a single COM object property.

Component Handle Expression

Every method or property reference must begin with a component handle expression that returns a component handle value. This is the syntax to specify a component handle expression:

SYNTAX

```
COMhandle [ : Method-Name-Reference | Property-Name-Reference ] ...
```

COMhandle

A component handle variable. (Note that the first element in a component handle expression must be a COM-HANDLE variable.)

Method-Name-Reference

Specifies a single COM object method or property that returns a component handle value. A component handle expression can chain as many method and property references as required to return the handle to a particular COM object.

Property-Name-Reference

Specifies a single COM object method or property that returns a component handle value. A component handle expression can chain as many method and property references as required to return the handle to a particular COM object.

NOTE: For the most efficient dispatch of multiple references to a particular COM object, assign the initial component handle expression for the COM object to a COM-HANDLE variable. Each reference to a given component handle expression incurs additional run-time overhead for each method or property referenced in the expression.

Method Name Reference

This is the syntax to specify a single COM object method:

SYNTAX

```
Method-Name
  (
    {     [ OUTPUT | INPUT-OUTPUT ]
             expression [ AS datatype ]
             [ BY-POINTER | BY-VARIANT-POINTER ]
        |  null-parameter
    }
    [ ,     [ OUTPUT | INPUT-OUTPUT ]
               expression [ AS datatype ]
               [ BY-POINTER | BY-VARIANT-POINTER ]
          |  null-parameter
    ] ...
  )
```

Method-name

The name of the COM object method. This name is not case sensitive. But by convention, Progress documentation shows COM object method names in mixed case.

expression

Any valid Progress expression that you can pass as a parameter to the method.

datatype

One of several data-type specifiers that the associated parameter might require. For more information on *datatype*, see Table 7–2. The remaining keyword options specify additional type and mode information for each parameter. The COM object defines the data types and numbers of parameters for a method.

null-parameter

Any amount of white space, indicating an optional parameter that you choose to omit. You can also pass variable numbers of parameters if the method supports it. For more information on specifying method parameter options, see the "Specifying Options For Properties and Method Parameters" section.

You can invoke a method in two ways:

- Include the appropriate method reference as part of a Progress expression. This expression can be on the right side of an assignment statement or in any other statement that accepts the expression. Methods invoked with this technique must return a value. For example, if YearDay( ) is a method that returns the number of days from the first of the year to a specified date, it can appear in the following bolded expressions:

```
DEFINE VARIABLE myObject AS COM-HANDLE
DEFINE VARIABLE day-of-year AS INTEGER.

/* ... Access COM object as myObject ... */

day-of-year = myObject:YearDay("05/01/1997").
DISPLAY myObject:YearDay(TODAY) " days have passed this year.".
```

- Specify the appropriate method reference as a statement, ignoring any return value. Methods invoked as a statement might require the NO-RETURN-VALUE option (shown previously in the Method Reference syntax). If SetDate( ) allows you to set a date for a COM object, you might invoke it in the following bolded statements:

```
DEFINE VARIABLE myObject AS COM-HANDLE
DEFINE VARIABLE day-of-year AS INTEGER.

/* ... Access COM object as myObject ... */

NO-RETURN-VALUE myObject:SetDate(1995,5,1). /* Set "May 1, 1995" */
myObject:SetDate(,5,1). /* Set "May 1" of current year */
```

Property Name Reference

This is the syntax to specify a single COM object property:

SYNTAX

```
Property-Name [ ( index [ , index ] ... ) ]
  [ AS datatype-specifier ]
```

Property-Name

The name of the COM object property. This name is not case sensitive. But by convention, Progress documentation shows COM object property names in mixed case.

index

Any expression that legally indexes the property and for the required number of dimensions. If necessary and if possible, Progress converts the data type of *index* to the COM data type expected by the property. Essentially, the syntax is the same as for a method reference with input parameters.

datatype-specifier

One of several data-type specifiers that the associated property might require when you set its value. For more information on *datatype*, see Table 7–2. For more information on specifying the AS *datatype* option and data type conversion for properties, see the "Specifying Options For Properties and Method Parameters" section.

You can reference a property in two ways:

- Read the property by specifying the appropriate property reference as part of a Progress expression. This expression can be on the right side of an assignment statement or in any other statement that accepts the expression, as in the following bolded expressions:

```
DEFINE VARIABLE myObject AS COM-HANDLE
DEFINE VARIABLE degF AS INTEGER.

/* ... Access COM object as myObject ... */

degF = myObject:TempFahrenheit.
DISPLAY "The current temperature is " myObject:TempFahrenheit " degF.".
```

- Write (set) the property by specifying the appropriate property reference on the left side of an assignment statement, as in the following bolded property references:

```
DEFINE VARIABLE myObject AS COM-HANDLE

/* ... Access initial COM object as myObject ... */

myObject:SavingsAcount("1000-24-369"):AccountName(1) = "John".
myObject:SavingsAcount("1000-24-369"):AccountName(2) = "Doe".
```

  Note the use of a chained component handle expression to reference a savings account object. AccountName is an indexed property of the SavingsAccount object that specifies a first and last name for the account.

Data-type Specifier

Progress supports a protocol to provide default mappings between native COM object method parameter and property values and the corresponding Progress data values. This mapping protocol supports many COM data types. You can override the default mapping by using a type specifier, as shown in Table 7–2. For more information on mapping data types shown in Table 7–2, see Appendix B, "COM Object Data Type Mapping."

Table 7–2: Data-type Specifiers For COM Object Methods and Properties

Data-type Specifier To Override the Default Mapping	Progress Data Type
CURRENCY	DECIMAL
ERROR-CODE	INTEGER
IUNKNOWN	COM-HANDLE
FLOAT	DECIMAL
SHORT	INTEGER
UNSIGNED-BYTE	INTEGER

Each of these data-type specifiers represents one of the base COM data types available for a method parameter or property. Each Progress data type represents the typical Progress data type that corresponds to the specified COM data type.

Data-type specifier options are necessary only for some method input parameters and property settings. Whether or not you must specify the AS *datatype* or any of the other type options (BY-POINTER or BY-VARIANT-POINTER) depends on the COM object method or property and the implementation of the COM object. You can also specify how a method parameter of any data type is passed using the mode options OUTPUT or INPUT-OUTPUT. Whether you use a mode option depends (in part) on how you plan to use the method parameter in your application. For more information on using data-type specifiers and mode options, see the "Specifying Options For Properties and Method Parameters" section and Appendix B, "COM Object Data Type Mapping."

7.3.2 Specifying Options For Properties and Method Parameters

Progress allows you to specify a variety of data-type specifier and mode options for passing COM object method parameters and setting COM object properties (see the "Property and Method Syntax" section). Data-type specifier options specify a data type mapping between COM data types and Progress data types; mode options specify how a method parameter is passed (whether for input or output). Thus:

- The data-type specifier options in Table 7–2 dictate COM data type conversions for passing method input parameters and setting properties that are different than the defaults. (For more information on COM data-type conversion, see Appendix B, "COM Object Data Type Mapping.")
- The type options BY-POINTER and BY-VARIANT-POINTER specify additional information for passing method parameters.
- The mode options OUTPUT and INPUT-OUTPUT specify how the method parameter is used.

One of the essential criteria that determines when and how you might have to use data-type specifier and mode options is the Type Library provided with a COM object implementation.

Understanding a COM Object Type Library

A *Type Library* contains definitions for a COM object's methods and properties. When a COM object provides a Type Library, Progress references it before dispatching the method or property in an attempt to convert each method parameter or property value to the required COM data type. If a Type Library is available, Progress tries to match the number and types of any parameters being passed into a method before dispatching the method to the COM object for execution.

If both data-type specifier options and Type Library definitions are provided, the data-type specifier options take precedence. For more information on how Progress matches Progress data items to COM object properties and method parameters, see Appendix B, "COM Object Data Type Mapping."

You can locate data type information that is stored in Type Libraries on-line using the Progress COM Object Viewer. For more information on Type Libraries and how to view their components, see the "Locating COM Object Information On Your System" section.

Using Data-type Specifier Options

Data-type specifier options allow you to be more specific about how to convert information from the Progress application into the data type expected by the COM object. These data-type specifiers override Progress default data type conversions for COM object properties and method parameters that have no Type Library support. For information on Progress default data conversions and how they are affected by data-type specifiers and Type Library support, see Appendix B, "COM Object Data Type Mapping."

Using Data-type Specifiers For Properties

You can include a data-type specifier option in a property reference on the left side of an assignment statement, when you set the property. You can use any data-type specifier from Table 7–2:

```
myObject:Money AS CURRENCY = 1000.9999.
```

Using Data-type Specifiers For Method Parameters

For a method parameter, you can also use a data-type specifier from Table 7–2. For COM objects that do not have Type Library definitions and yet require that the data type of the parameter be passed properly, you must specify the data type for the method call to succeed:

```
myObject:SaveMoney(1000.9999 AS CURRENCY).
```

In addition, you can use the BY-POINTER or BY-VARIANT-POINTER type option to indicate that the parameter is to be passed as a pointer:

```
myObject:SaveMoneyPtr(1000.9999 AS CURRENCY BY-POINTER).
```

A *pointer* is a value that contains the memory location of the data item you are referencing. BY-POINTER specifies a pointer to the data item value. BY-VARIANT-POINTER specifies a Variant that contains a pointer to another Variant that stores the actual value.

NOTE: A Variant is a self-describing COM data type. It contains both the data and an indication of its effective data type.

Both the BY-POINTER and BY-VARIANT-POINTER options have no affect on the value of the parameter. They only affect how the data is packaged when the method is dispatched to the COM object. The type option to use, if any, is determined by the method implementation. You can determine the type options required for each parameter from the Progress COM Object Viewer (see the "Locating COM Object Information On Your System" section).

Using Mode Options For Method Parameters

The default mode for a method parameter is input. An input parameter passes a value to the method but does not return a value from the method. Thus, an input parameter can be a database field, an expression, or a variable.

CAUTION: Do not use the INPUT keyword as a mode option because, for a parameter, Progress might interpret this as the screen value of a widget.

The mode option OUTPUT or INPUT-OUTPUT specifies a parameter that returns a value from the method. (An INPUT-OUTPUT parameter also passes a value to the method.) This means that the value of any passed variable can change after the method call returns. You can only specify the OUTPUT or INPUT-OUTPUT options with a variable as the parameter (as opposed to a database field or an expression):

```
DEFINE VARIABLE MyWallet AS DECIMAL.
myObject:WithdrawMoneyPtr(OUTPUT MyWallet AS CURRENCY).
```

NOTE: The OUTPUT or INPUT-OUTPUT option forces the parameter to be passed as a pointer and explicitly specifies that a value be returned to your application. Thus, if you use a mode option, you do not need to use the BY-POINTER type option because the type option is redundant. However, the BY-POINTER type option, by itself, does not return a value to your program. You must use a mode option or Progress does not allow the method to return a value in the parameter.

Note that Progress does not use Type Library information to determine the parameter mode. This prevents the COM object from updating a variable that you do not expect or want to change. Thus, if the COM object ordinarily changes the value of a particular parameter, you can prevent any variable you pass from having its value changed by omitting any mode option on the parameter.

7.3.3 Restrictions On Property and Method References

Progress supports most features necessary to reference COM object properties and methods. However, there are some restrictions:

- Parentheses are required for method calls, whether or not they take parameters.
- There is no support for named (keyword) parameters.
- There is no support for default properties or methods.
- There is no array support for method parameters.

Parentheses On Method Calls

In general, you must specify all method references with parentheses. Although some COM objects accept a method call referenced without parentheses, Progress is not aware that such a reference is a method call. Without parentheses, the compiler interprets the reference as a property rather than as a method. This can cause unpredictable results for the method call.

Named Method Parameters

The COM standard allows named method parameters that you can specify in any order for a method call. Progress does not support named parameters in any form. The first line of this example shows an illegal named parameter, where the Filename parameter is passed to the SaveAs method on an Excel Workbook handle:

```
Excel-Workbook-hdl:SaveAs (Filename = "x.xls"). /* Illegal syntax */
Excel-Workbook-hdl:SaveAs ("x.xls").            /* Legal syntax */
```

The second line shows the form that Progress accepts. In general, whether or not you omit optional parameters, you must pass all method parameters in the correct parenthesized order.

Default Properties and Methods

The COM standard allows for default properties and methods. For any COM object, the default property or method is invoked when you reference only the COM object handle. For example, Item is the default (indexed) property for collection objects. However, Progress requires that you specify all properties and methods you want to reference, whether or not they are defaults for the COM object:

```
DEFINE VARIABLE hCollection AS COM-HANDLE.

/* ... Set hCollection to a collection COM object ... */

hCollection:Item(1). /* This is legal in Progress. */
hCollection(1).     /* This is not legal in Progress. */
```

Array Method Parameters

Progress does not support array method parameters. This example, that returns the 12 monthly average temperatures for the year ending today, is illegal:

```
DEFINE VARIABLE vAvgTemp AS DECIMAL EXTENT 12. /* 12 Monthly DegF */

/* ... Instantiate an Annual Weather History Object ... */

chAnnual:MonthlyAvgDegF(TODAY, OUTPUT vAvgTemp).
```

A Progress array variable can be passed only one extent at a time.

NOTE: An array of bytes can be passed in a RAW.

7.4 Managing COM Objects In an Application

Progress provides a number of mechanisms in the 4GL to help manage COM objects in an application. Some of these mechanisms derive from general 4GL constructs previously available, while others are added to the 4GL just for COM object support. Together, they support the following application tasks:

- Component handle validation
- Font and color management
- ActiveX collection navigation
- Resource management
- Error handling

7.4.1 Validating Component Handles

If you reference a COM-HANDLE variable whose value does not point to a valid COM object, Progress returns an error indicating that an action was performed on an invalid COM-HANDLE. To protect against this error, use the Progress VALID-HANDLE function to determine if the COM-HANDLE variable contains a valid value before using it in any other Progress statement. As with widget handles, the VALID-HANDLE function returns TRUE if the component handle is valid.

Note that you cannot use the VALID-HANDLE function to verify that a component handle value points to a particular COM object. COM-HANDLE values might be reused within an application when the COM objects they point to are no longer available. For more information, see the "Managing COM Object Resources" section.

Also, this function only indicates that a component handle is invalid from some action (or inaction) of the Progress application. It does not show as invalid a COM handle that a user might have manually closed, for example, an Automation Server application that provided the COM object.

7.4.2 Managing COM Object Font and Color Settings

You can manage both fonts and colors for COM objects from the 4GL. However, the 4GL provides more direct support for color than for font management.

Managing Fonts

Most Automation Servers or ActiveX controls that support font manipulation provide an associated Font object to manage font changes. As such, you can use Font object properties and methods to read or set font values for the COM object. For information on font support, see the documentation on your Automation Server or ActiveX control. Otherwise, there is no mapping between COM object font settings and the Progress font information maintained by the FONT-TABLE system handle.

Managing Colors

COM objects accept color specifications in the form of an RGB (Red/Green/Blue) integer value. However, Progress widgets accept color specifications in the form of an integer index into a color table managed by the COLOR-TABLE system handle. To support color management for COM objects, Progress provides techniques that work with or without the Progress color table.

Progress provides three ways to obtain a color value to set colors for a COM object:

- Use the value from the property or method of another COM object.
- Use the GET-RGB-VALUE( ) method of the COLOR-TABLE system handle.
- Use the RGB-VALUE function.

To use a color value from another COM object, simply assign the color value returned by one of its properties or methods to a color property or as a method parameter of your Automation object or ActiveX control.

To use the GET-RGB-VALUE( ) method, pass it an index to a color stored in the Progress color table and the method returns an integer that represents the RGB value of the specified color. You can then assign this value to a COM object color property:

```
myObject:BorderColor = COLOR-TABLE:GET-RGB-VALUE(5).
```

This example assigns the RGB value of color number 5 from the color table to the BorderColor property of myObject.

To use the RGB-VALUE function, you pass the function three color values between 0 and 255 and it returns a single RGB value that represents the color:

```
myObject:BorderColor = RGB-VALUE(127, 255, 75). /* Red, Green, Blue */
```

7.4.3 Navigating ActiveX Collections

ActiveX collections are COM objects that reference multiple instances of a particular class of COM object. Progress supports collection navigation by allowing you to access a COM object through the indexed Item property of the collection object:

```
DEFINE VARIABLE i AS INTEGER.
DEFINE VARIABLE ExcelApp AS COM-HANDLE.

/* Instantiate Automation object for Excel.Application in ExcelApp */

CREATE "Excel.Application" ExcelApp.
  .      .
  .      .
  .      .
REPEAT i = 1 TO ExcelApp:Sheets:Count():
    ExcelApp:Sheets:Item(i):Name = "ABC" + STRING(i).
END.
```

In this example, the 4GL loops through all the worksheet objects in the Excel Application collection. Sheets is the collection and Item is the indexed property that returns the component handle to each Sheet object. The code uses the index (i) to loop through the total number of Sheet objects in the collection (ExcelApp:Sheets:Count( )), and assigns a unique name to each one ("ABC1", "ABC2", and so on).

NOTE: Collections are often named as a plural of the object class that they index. Thus, in this example, Sheets is the collection class and Sheet is the object class whose instances are indexed by the Item property of Sheets.

7.4.4 Managing COM Object Resources

When working with COM objects, especially Automation objects, it can be very easy to instantiate many object instances (for example, when searching many objects in a collection). This accumulation of COM objects can impose a burden on system resources. To alleviate this burden, Progress provides the RELEASE OBJECT statement. This statement releases the COM object associated with the specified component handle, making it available for garbage collection if no other COM object has a reference to it. In general, it is good practice to release any COM handles that you no longer need for your application. It is only necessary to release COM objects that have been assigned to a component-handle variable.

Releases and Deletes

Note that you release COM objects, but delete widgets. A release is different from a delete because, by convention, a COM object stays around until there are no other references to it. In the case of an ActiveX control, the control-frame is a COM object that references the control (see Chapter 1, “Introduction.”). If you set a component handle variable to the control, this is a second reference. However, no matter how many component handle variables you set to the same control, this represents a single reference from Progress. (Progress takes care of this for you.) If you release the control through a component handle while its control-frame is still instantiated, the control remains instantiated because the control-frame COM object still references the control. In this case, you must delete the control-frame widget to finally free the ActiveX control. In a similar way, multiple COM object references can also keep an Automation object and its resources tied up longer than necessary.

Release Strategy

In general, to maximize the reusability of COM object resources, always release a COM object when you no longer need it. Because of references between COM objects, the status of references to a particular COM object might not always be obvious (especially for Automation objects). By releasing COM objects as soon as you know they are not needed in an application, you have the best chance of ensuring that their resources are available for reuse as soon as possible.

Releases and Component Handles

When you release a COM object, this invalidates every component handle that references the same object. Any further attempt to use an invalidated component handle results in an error indicating that an invalid action was performed on an invalid COM-HANDLE value.

On the other hand, if you instantiate a different COM object after releasing the first one, a previously invalidated component handle might point to the new COM object. The component handle might be reused because the object server might reuse memory left over from the released COM object. In this case, the component handle might be seen as valid. However, the object that the component handle references is different, so this might result in errors or successful method calls with results that are invalid for your application.

In general, to avoid errors and confusion with invalidated COM handles, it is a good practice to set any COM-HANDLE variables to the unknown value (?) once you have released it.

NOTE: Progress maintains no relationship between one component handle and another, such as when you derive one component handle from a property value on another component handle. The two component handles are completely independent of each other. This is different from the parent/child relationships that some COM objects (especially Automation objects) might maintain among themselves. In this case, invoking the right method on a "parent" COM object might well trigger the release of many other "related" COM objects. For information on any such cascading object relationships, see the documentation on a particular Automation Server or ActiveX control.

7.4.5 Error Handling

Progress traps all COM object errors caused by property and method references. Progress formats the error information into a Progress error message that includes the hexadecimal code of the COM object error and explanatory text. If you specify the NO-ERROR option on a statement that references a COM object property or method, Progress stores the error information in the ERROR-STATUS handle:

```
DEFINE VARIABLE chMyObject AS COM-OBJECT.
DEFINE VARIABLE i AS INTEGER.

/* ... Instantiate COM Object with chMyObject ... */

chMyObject:Method(10) NO-ERROR.
IF ERROR-STATUS:NUM-MESSAGES > 0 THEN
   DO i = 1 TO ERROR-STATUS:NUM-MESSAGES:
      MESSAGE ERROR-STATUS:GET-MESSAGE(i).
   END.
```

NOTE: Some types of Progress statements treat any errors as warnings even if the method or property reference results in a serious error. For warnings, Progress does not set the ERROR-STATUS:ERROR attribute. Thus, to detect that any exception (warning or error) occurred, you must check ERROR-STATUS:NUM-MESSAGES for a value greater than zero, as in the example.

You cannot otherwise access COM object error information directly unless a method includes this information in its return value or in an output parameter. This means you cannot reliably respond to a particular COM object error code. You can only tell that an error occurred and search the message text for a set of likely error codes.

7.5 Locating COM Object Information On Your System

You can view the COM objects provided with any Automation Server or ActiveX control that is installed on your system by using the Progress COM Object Viewer. This tool parses the Type Library installed with a selected Automation Server or ActiveX control to provide three lists:

- Automation objects that are *createable* (usable in the CREATE Automation Object statement) or ActiveX controls that are selectable as OCXs in the AppBuilder
- All COM objects supported by the selected Automation Server or ActiveX control
- The properties, methods, and events for any selected COM object

7.5.1 Using the COM Object Viewer

The general steps to use the COM Object Viewer include:

1 ♦ Run the Viewer in one of the following ways:

   a) PRO*Tools in the Progress Application Development Environment (ADE)

   b) Windows Explorer

   c) The command line

2 ♦ Open the Type Library for an Automation Server or ActiveX control.

3 ♦ Locate objects in the Viewer.

4 ♦ Review, cut, and paste in your application any available syntax to use an object.

7.5.2 Running the COM Object Viewer

To execute the Viewer from PRO*Tools:

1 ♦ Choose PRO*Tools from the Tools menu in any ADE window.

2 ♦ Click the Progress COM Object Viewer button in the PRO*Tools toolbar:

Progress COM Object Viewer button

To execute the Viewer from Windows Explorer, navigate to your `%DLC%\bin` installation directory and double-click the `proobjvw.exe` icon.

To execute the Viewer from the command line, open an MS-DOS Prompt window and enter `proobjvw.exe` at the prompt.

7.5.3 Accessing Type Libraries

Type Libraries describe most COM objects, including both Automation objects and ActiveX controls. A Type Library can exist in one of the following forms:

- A separate file with the extension `.tlb` or `.olb`. This file is usually in the same directory as the main binary file for the Automation Server or ActiveX control.
- Part of the main binary file for the Automation Server or ActiveX control, generally with the extension `.exe`, `.dll`, or `.ocx`. This is often the form provided with ActiveX controls.

7.5.4 Locating Objects In the Viewer

The top listbox label in the Viewer changes depending on whether you open an Automation Server Type Library or an ActiveX control Type Library. For Automation Servers, the top listbox is Automation Objects. For ActiveX controls it is Controls.

Automation Objects

For Automation Servers, the Viewer displays all createable Automation objects in the Automation Objects listbox, as shown in Figure 7–1.

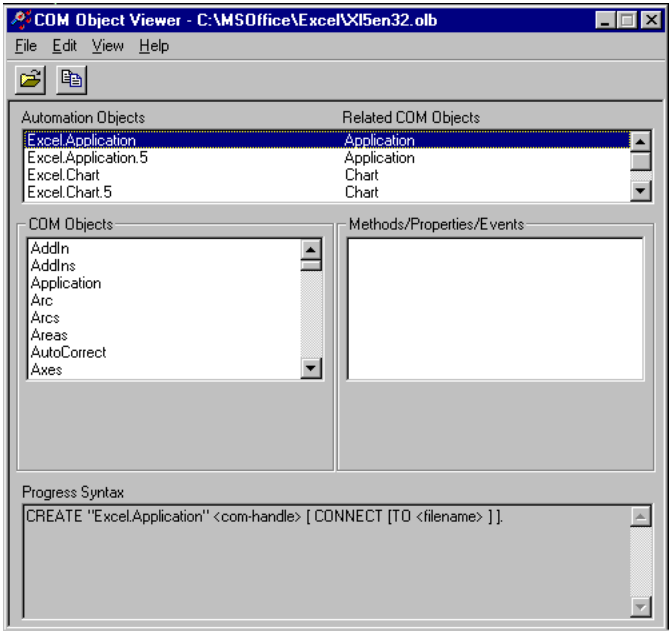


Figure 7–1: Automation Objects In the COM Object Viewer

In COM, a createable Automation object has an identifier known as a Program Identifier (ProgID in the registry). This identifier is the expression that you use to identify the Automation object in the 4GL CREATE Automation Object statement (see Chapter 8, "ActiveX Automation Support."). The Automation Objects listbox lists the ProgID of each createable Automation object followed by the corresponding COM object (Related COM Objects).

When you select an item in the Automation Objects listbox, a Progress Syntax editbox at the bottom of the window shows sample 4GL syntax for creating it. You can cut and paste this syntax into a 4GL procedure.

The Viewer lists all COM objects that are available from the Automation Server to an Automation Controller, like Progress, in the COM Objects listbox. In general, only a small number of Automation objects are createable. You then use the properties and methods on these COM objects to access the other COM objects listed for the Server. You cannot determine the relationship among COM objects from the Progress COM Object Viewer tool. For more information on this, see the documentation provided with each Automation Server.

ActiveX Controls

For ActiveX controls, the Viewer displays the name of the control in the Controls listbox that is selectable as an OCX in the AppBuilder, as shown in Figure 7–2.

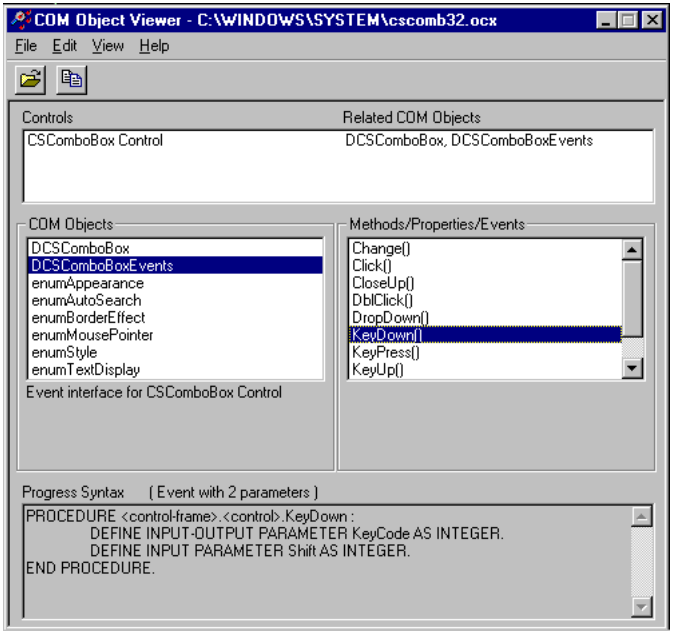


Figure 7–2: ActiveX Controls In the COM Object Viewer

The listed control name is the OCX name (not the control-frame name) that the AppBuilder displays for the control when you select and insert it in a design window. This name is followed by the names of corresponding COM objects (Related COM Objects).

When you select an item in the Controls listbox (not shown selected), no syntax appears in the Progress Syntax editbox at the bottom of the window. This is because the AppBuilder generates all required syntax for creating an ActiveX control in your application at run time.

The viewer lists all COM objects that are available to Progress with this control in the COM Objects listbox.

In general, only one listed object is available in the AppBuilder at design time, the ActiveX control, itself. You then use the properties, methods, and events on the ActiveX control to access the other COM objects listed in the COM Objects listbox at run time. (For more information on ActiveX control events, see Chapter 9, "ActiveX Control Support.") You cannot determine the relationship among COM objects from the Progress COM Object Viewer tool. For more information on this, see the documentation provided with each ActiveX control.

7.5.5 Viewing Methods, Properties, and Events

When you select a COM object in the COM Objects listbox, a Methods/Properties/Events listbox shows the methods, properties, and events of the COM object in alphabetical order. Methods and events appear with a pair of parentheses following the name. Properties for constant COM objects appear as a set of constant values.

When you select an item in the Methods/Properties/Events listbox, the Progress Syntax editbox shows sample 4GL syntax for using it. You can cut and paste this syntax into a 4GL procedure. Figure 7–2 shows the OCX event procedure syntax for the CSComboBox KeyDown event.

The tool also displays information on the method, property, or event above the Progress Syntax editbox, including:

- The number of parameters for the method, indexed property, or event
- Whether the property is read-only
- Whether a constant value is being displayed

With the sample 4GL syntax, the tool displays any important information on parameters and return data type, including:

- Methods that do not return a value (called with NO-RETURN-VALUE)
- The return data type of a method that does return a value (shown prefixed to the name of the sample variable on the left side of the assignment)
- The data type of the parameters (shown prefixed to the name of each sample variable for a parameter)
- The mode of the parameter (OUTPUT or INPUT-OUTPUT)
- The value of a constant
- Optional portions of the syntax in brackets ([...])
- PROCEDURE definitions for events, including any parameters.

8

ActiveX Automation Support

ActiveX Automation allows one application (the *Automation Controller*) to drive another application (the *Automation Server*) through COM objects (*Automation objects*) provided by the Automation Server. An Automation object thus represents both a package of Server functionality and a point of connection from the Automation Controller to the Automation Server. This functionality can include many capabilities from exchange of data between the two applications to almost total control of the Server application by the Controller.

Progress supports ActiveX Automation as an Automation Controller. This allows you to write a 4GL application that connects to and drives Automation objects in an Automation Server. You do this by creating a connection to the Automation object and referencing its properties and methods.

This chapter describes how to build a 4GL application that functions as an Automation Controller, featuring a working example. It relies on Progress support for COM object access in the 4GL. For information on Progress support for COM objects, see Chapter 7, "Using COM Objects In the 4GL."

This chapter contains the following sections:

8.1 Requirements For Doing Automation

The main requirement for implementing Automation in an application is that the Automation Server is correctly installed on the system where you develop and deploy the application.

The basic steps to write an Automation application include:

1 ♦ Access an Automation Server by creating connections to one or more createable Automation objects provided by the Server (using the CREATE Automation Object statement).

2 ♦ Access data and functionality on the Server and instantiate additional Automation objects as needed, through the properties, methods, and events of the created Automation objects.

3 ♦ Enable events for the Automation object, if desired.

4 ♦ Release each Automation object as your application no longer needs them.

5 ♦ Repeat Steps 1 through 3 for as many Automation Servers as your application requires.

The possible combinations of Servers and Automation objects used by your application are limited only by the available resources and implementation of the Automation Servers.

Note that Automation objects can be organized into hierarchies in an Automation Server, with one or more top-level Automation objects providing access to the rest. Top-level objects represent the application for the Server and are always createable. Other Automation objects in the Server can also be createable. The relationship of each Automation object in the hierarchy affects how you can instantiate it. For more information on the relationship among Automation objects, see the documentation on their Automation Server.

8.2 Accessing Automation Servers

The CREATE Automation Object statement provides four basic connection options to access an Automation Server. (For more information, see the entry for the CREATE Automation Object Statement in the *Progress Language Reference*.) Each option handles different connection requirements, depending on the Automation object implementation and the execution status of the Automation Server. Any Automation object flagged in the registry with REGCLS_MULTIPLEUSE launches only a single instance of the Server that handles all subsequent object instantiation requests. Any Automation object flagged in the registry with REGCLS_SINGLEUSE launches a Server dedicated to that object instance. Any subsequent Automation object instantiated for the same Server launches a new instance of the Server dedicated to that object.

8.2.1 Option 1: Instantiate Automation Object by Name

This option creates a connection to a new instance of a specified Automation object, launching the Server if necessary. For top-level Automation objects (such as Excel.Application), this option usually launches a new instance of the Server, unlike for lower-level objects (such as Excel.Sheet, which use the instance created by a top-level object):

```
DEFINE VARIABLE hExcelObject AS COM-HANDLE
CREATE "Excel.Application" hExcelObject.
```

Figure 8–1 summarizes the basic logic for this option.

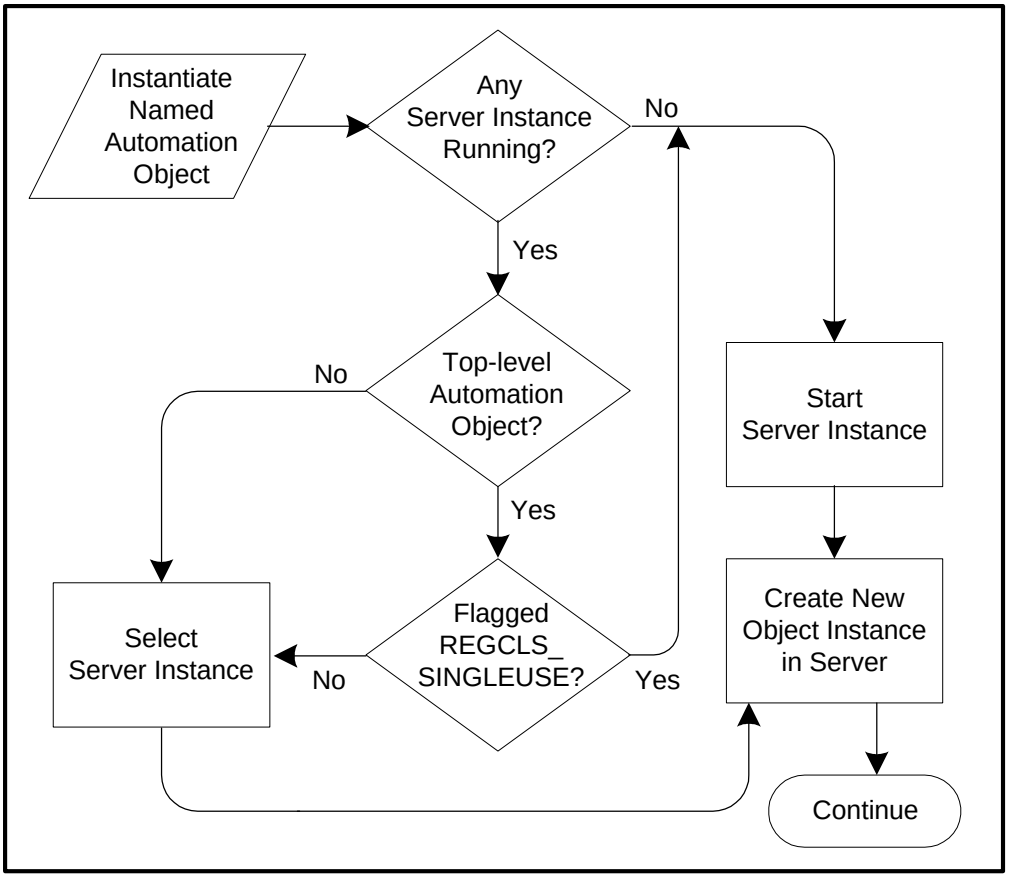


Figure 8–1: Automation Connection Option 1

NOTE: This option is equivalent to the Visual Basic CreateObject(*class*) or GetObject("",*class*) function call.

8.2.2 Option 2: Connect To Top-level Named Automation Object

This option creates a connection to an existing instance of a top-level Automation object (such as "Excel.Application"). This does not work with lower-level objects (such as "Excel.Sheet"), and fails if the Server is not already running:

```
DEFINE VARIABLE hExcelObject AS COM-HANDLE
CREATE "Excel.Application" hExcelObject CONNECT.
```

Figure 8–2 summarizes the basic logic for this option.

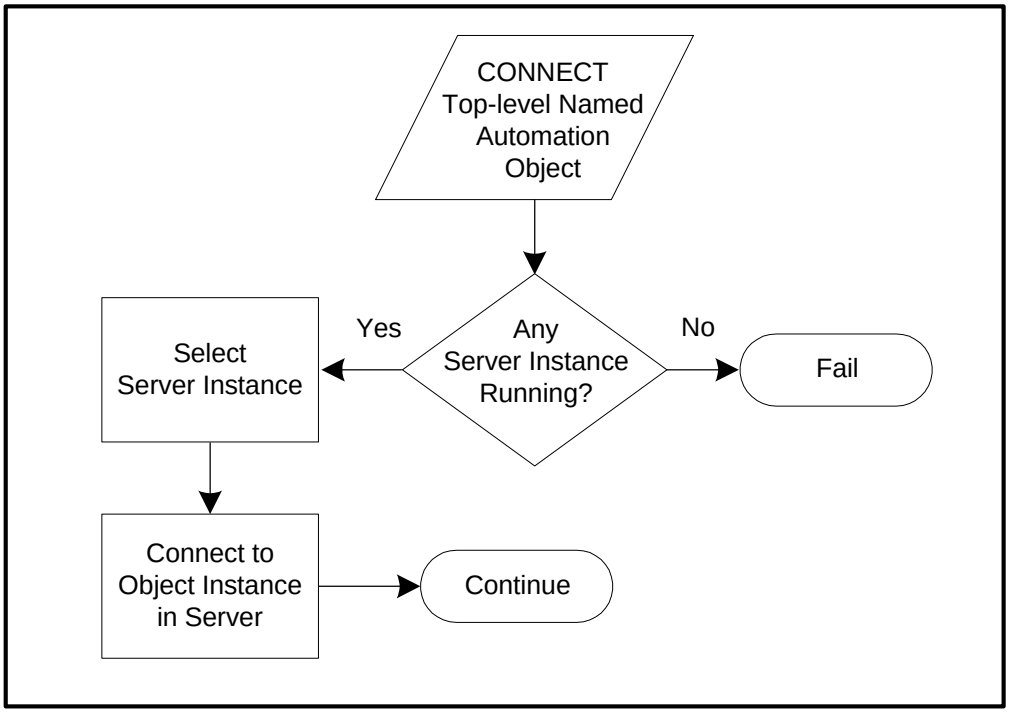


Figure 8–2: Automation Connection Option 2

NOTE: This option is equivalent to the Visual Basic GetObject(,*class*) function call.

8.2.3 Option 3: Connect To Or Instantiate a Named Automation Object and File

This option creates a connection to a new or existing instance of the specified Automation object and opens the specified file. If the file is not already open, it is opened. If the pathname for the specified file is invalid or unrecognizable by the Server, this connection option fails. In this example, the file `\WorkSheets\Xplan.xls` is opened in the new or existing instance of the "Excel.Sheet" object:

```
DEFINE VARIABLE hExcelObject AS COM-HANDLE
CREATE "Excel.Sheet" hExcelObject CONNECT TO "\WorkSheets\Xplan.xls".
```

Figure 8–3 summarizes the basic logic for this option (ignoring the listed error conditions).

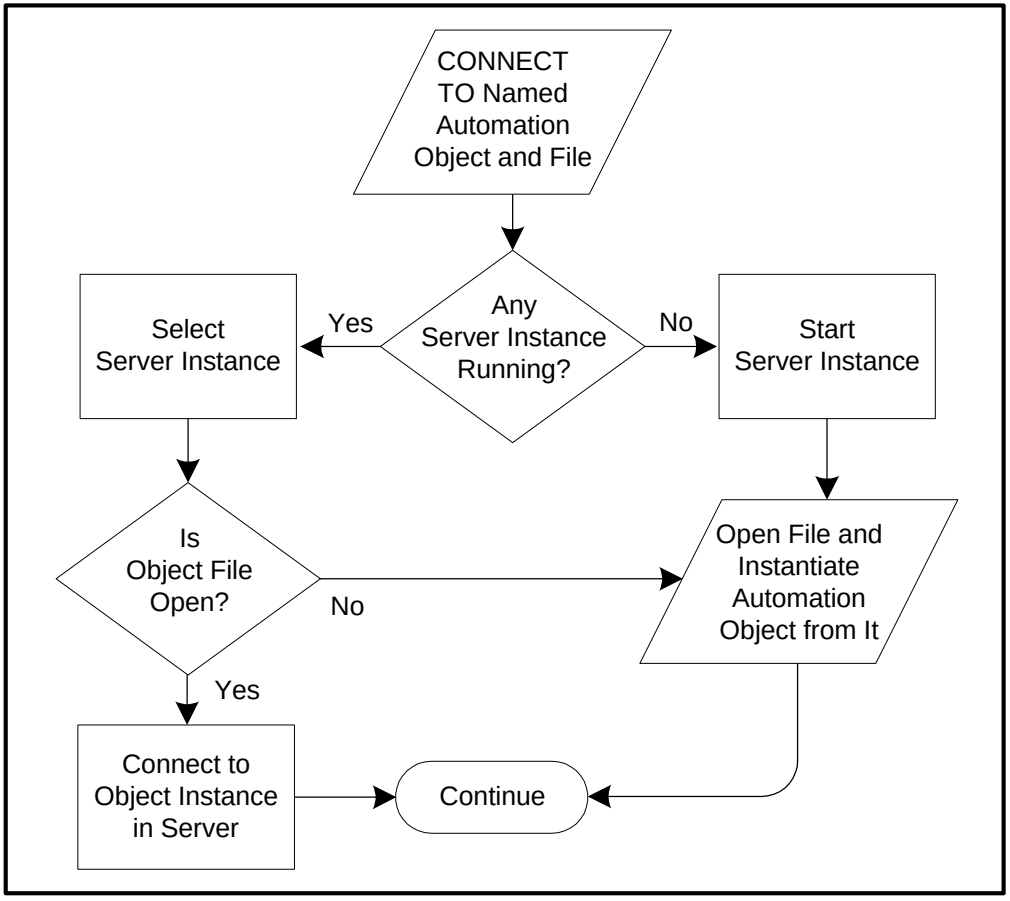


Figure 8–3: Automation Connection Option 3

NOTE: This option is equivalent to the Visual Basic GetObject(*pathname*,*class*) function call.

8.2.4 Option 4: Connect To Or Instantiate Implied Automation Object and File

This option creates a connection to a new or existing instance of the Automation object implicitly defined by the specified file. This option identifies the Automation object and its Server from the specified filename extension, as defined in the registry. If the file is not already open in the Automation Object, it will be opened. In this example, the `.xls` extension indicates that the object instance is a Sheet object provided by the Excel Automation Server:

```
DEFINE VARIABLE hExcelObject AS COM-HANDLE
CREATE "" hExcelObject CONNECT TO "\WorkSheets\Xplan.xls".
```

Figure 8–4 summarizes the basic logic for this option (ignoring the listed error conditions):

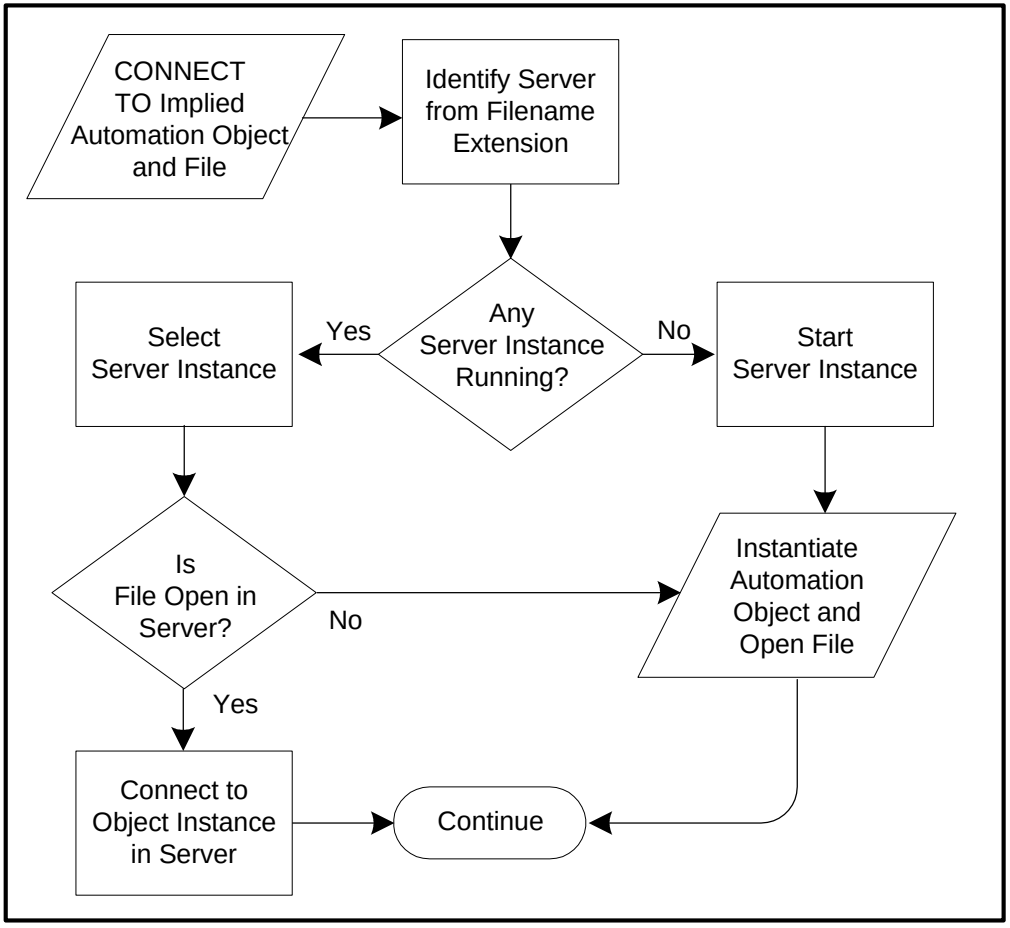


Figure 8–4: Automation Connection Option 4

NOTE: This option is equivalent to the Visual Basic GetObject(*pathname*) function call.

8.3 Managing Automation Objects

Progress provides the RELEASE OBJECT statement to release Automation objects that your application no longer requires. Efficient use of resources requires that you actively manage the Automation objects you instantiate in your application. Each object remains active until there are no remaining references from the Server or other Automation Controllers and one of the following events occurs:

- The Progress session terminates.
- You actively release the Automation object.

To efficiently manage Automation, you must release all Automation objects that you instantiate, either directly (through the CREATE Automation Object statement) or indirectly (through properties and methods of other Automation objects). There is no association between Automation objects that automatically propagates the release of one from the release of another.

In general, always release an Automation object when you are certain no other functionality in your application requires it. Note that if more than one component handle variable references the same Automation object, releasing the object with one handle invalidates them all.

CAUTION: To avoid misleading errors while developing your application, set all equivalent component handles to the unknown value (?) after releasing Automation objects on any one of them.

For more information on managing Automation objects, see the sections on COM object management in Chapter 7, “Using COM Objects In the 4GL.”

8.4 Automation Event Support

Progress supports event notification for ActiveX Automation objects with a built-in method that enables event notification for that specific component handle.

NOTE: This method is invalid when the component handle represents a control, since events work for controls automatically.

The following syntax describes the ENABLE-EVENTS method for ActiveX Automation objects:

SYNTAX

```
ENABLE-EVENTS ( event-proc-prefix )
```

event-proc-prefix

A CHARACTER expression representing a string prepended to every event received. The resulting string is the name of the internal procedure Progress runs in response to an event notification. During an event notification, all running procedures and all persistent procedures are searched to find a procedure with the name matching *event-proc-prefix.eventname* (for example, ExcelWB.SelectionChanged).

For more information on managing Automation objects, see the sections on COM object management in Chapter 7, "Using COM Objects In the 4GL."

For information on ActiveX control event management, see Chapter 9, "ActiveX Control Support."

8.5 Example Automation Applications

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

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

The following procedure listing shows `oleauto.p`. The bolded code shows the five component handles and where they are used to instantiate and release Automation objects. Only the Excel Application object is instantiated with the CREATE Automation Object statement. The rest are instantiated from methods of the Application object and its subordinate objects.

As the comments indicate, this procedure starts Excel, generates the graph from the sports database, and exits leaving Excel and the graph open on your Windows desktop. Thus, this application really functions as a startup file for Excel and releases all of its instantiated Automation objects just prior to terminating. The objects that Excel requires remain instantiated:

oleauto.p *(1 of 2)*

```
/*
 * This sample extracts data from a Progress database
 * and graphs the information using the Automation Objects
 * from the Excel server in Microsoft Office.
 * You must connect to a sports database before running this.
 * This sample program leaves Excel open.  You should close it manually
 * when the program completes.
 */

DEFINE VARIABLE chExcelApplication      AS COM-HANDLE.
DEFINE VARIABLE chWorkbook              AS COM-HANDLE.
DEFINE VARIABLE chWorksheet             AS COM-HANDLE.
DEFINE VARIABLE chChart                 AS COM-HANDLE.
DEFINE VARIABLE chWorksheetRange        AS COM-HANDLE.
DEFINE VARIABLE iCount                  AS INTEGER.
DEFINE VARIABLE iIndex                  AS INTEGER.
DEFINE VARIABLE iTotalNumberOfOrders    AS INTEGER.
DEFINE VARIABLE iMonth                  AS INTEGER.
DEFINE VARIABLE dAnnualQuota            AS DECIMAL.
DEFINE VARIABLE dTotalSalesAmount       AS DECIMAL.
DEFINE VARIABLE iColumn                 AS INTEGER INITIAL 1.
DEFINE VARIABLE cColumn                 AS CHARACTER.
DEFINE VARIABLE cRange                  AS CHARACTER.

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

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

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

/* get the active Worksheet */
chWorkSheet = chExcelApplication:Sheets:Item(1).

/* set the column names for the Worksheet */
chWorkSheet:Columns("A"):ColumnWidth = 18.
chWorkSheet:Columns("B"):ColumnWidth = 12.
chWorkSheet:Columns("C"):ColumnWidth = 12.
chWorkSheet:Range("A1:C1"):Font:Bold = TRUE.
chWorkSheet:Range("A1"):Value = "SalesRep".
chWorkSheet:Range("B1"):Value = "Total Sales".
chWorkSheet:Range("C1"):Value = "Annual Quota".

/* Iterate through the salesrep table and populate
   the Worksheet appropriately */
```

oleauto.p *(2 of 2)*

```
FOR EACH salesrep:
    dAnnualQuota = 0.
    iTotalNumberOfOrders = 0.
    dTotalSalesAmount = 0.
    iColumn = iColumn + 1.
    FOR EACH order OF salesrep:
        iTotalNumberOfOrders = iTotalNumberOfOrders + 1.
        FIND invoice WHERE invoice.order-num = Order.order-num NO-ERROR.
        IF AVAILABLE invoice THEN
            dTotalSalesAmount = dTotalSalesAmount + invoice.amount.
    END.

    DO iMonth = 1 TO 12:
        dAnnualQuota = dAnnualQuota + salesrep.month-quota[iMonth].
    END.

    cColumn = STRING(iColumn).
    cRange = "A" + cColumn.
    chWorkSheet:Range(cRange):Value = salesrep.rep-name.
    cRange = "B" + cColumn.
    chWorkSheet:Range(cRange):Value = dTotalSalesAmount.
    cRange = "C" + cColumn.
    chWorkSheet:Range(cRange):Value = dAnnualQuota.
END.

chWorkSheet:Range("B2:C10"):Select().
chExcelApplication:Selection:Style = "Currency".

/* create embedded chart using the data in the Worksheet */
chWorksheetRange = chWorksheet:Range("A1:C10").
chWorksheet:ChartObjects:Add(10,150,425,300):Activate.
chExcelApplication:ActiveChart:ChartWizard(chWorksheetRange, 3, 1, 2, 1, 1,
TRUE, "1996 Sales Figures", "Sales Person", "Annual Sales").

/* create chart using the data in the Worksheet */
chChart=chExcelApplication:Charts:Add().
chChart:Name = "Test Chart".
chChart:Type = 11.

/* release com-handles */
RELEASE OBJECT chExcelApplication.
RELEASE OBJECT chWorkbook.
RELEASE OBJECT chWorksheet.
RELEASE OBJECT chChart.
RELEASE OBJECT chWorksheetRange.
```

9

ActiveX Control Support

An *ActiveX control (OCX)* is a reusable component built on the Microsoft Component Object Model (COM) that allows you to extend the user interface and functionality of a Progress application. Some controls are similar to 4GL widgets, such as combo boxes and radio sets. There are a wide variety of additional user interface controls that are not available as built-in 4GL widgets, such as spin buttons, various dialogs, meters, and picture controls. There are also nongraphical controls for such tasks as communications and time keeping that have no user interface.

This chapter describes the mechanics of working with ActiveX controls in the 4GL. It does not fully explain how to use the Progress AppBuilder to incorporate ActiveX controls in an application. For information on using the AppBuilder to work with ActiveX controls, see the *Progress AppBuilder Developer's Guide*. The information in this chapter relies on Progress support for COM objects. For information on COM objects in Progress, see Chapter 7, "Using COM Objects In the 4GL."

This chapter contains the following sections:

9.1 How Progress Supports ActiveX Controls

In general, ActiveX control technology is a more powerful, flexible, and robust replacement for the VBX control standard. To comply with the COM standard, an ActiveX control instance must be placed in a *control container* that handles events and specific user-interface functionality for the control. Progress supports this standard with the control-frame widget, a field-level widget and an associated control-frame COM object (which provides the actual control container support). For an overview of ActiveX controls, how they interact with Progress, and the requirements for using them in Progress, see Chapter 1, "Introduction."

You can extend a Progress application with numerous commercially available ActiveX controls. In Progress, you must use the AppBuilder at design time to create instances of ActiveX controls in your application. You can then write 4GL code to reference ActiveX control properties and methods and to define event handlers that respond to ActiveX control events.

9.1.1 An Example ActiveX Control In Progress

Figure 9–1 shows a spin button control (the large left- and right-arrow buttons) implemented in the example application, `e-ocx1.w`. This spin button control overlays a portion of the frame space defined by a Progress control-frame widget.

NOTE: This spin button control is one of three ActiveX controls provided with your Progress installation. The other two are combo box and timer controls. For more information on these controls, see the "ActiveX Controls and Examples Installed With Progress" section.

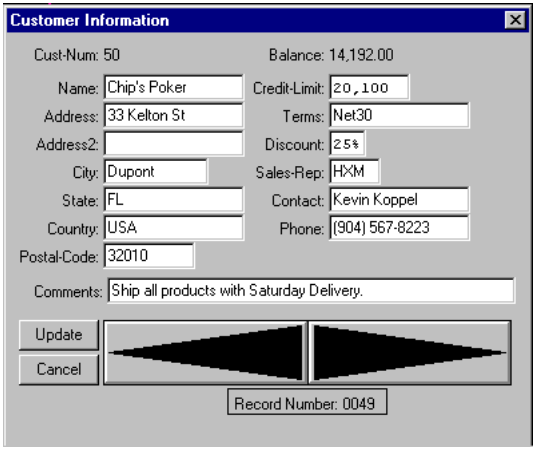


Figure 9–1: ActiveX Control Example

In this example, the control-frame (and thus, the control itself) has been largely stretched in the horizontal direction and slightly stretched in the vertical to present a convenient access point in the dialog box. A spin button control is generally designed to provide incremental and decremental values, like a slider. This application uses the events generated by spin button value changes to scan back and forth through a list of Customer records in the sports database. It uses the incremental and decremental control values to maintain a record count (Record Number field). Later sections show how the application handles these spin button control events and values in the 4GL.

9.1.2 How Progress Encapsulates an ActiveX Control

Progress encapsulates an ActiveX control using the control-frame. The control-frame provides the basic interface between the ActiveX control and the 4GL through its two separate but related objects, the control-frame widget and the control-frame COM object. (For an overview of the relationship between ActiveX controls and the control-frame, see Chapter 1, "Introduction.")

Control-frame Widget

The control-frame widget is a field-level widget, and as such it establishes the relationship between the ActiveX control and other Progress widgets in the user interface. Thus, it is the control-frame widget attributes (like ROW and TAB-POSITION) that maintain the relationship between the ActiveX control and other field-level widgets. For example, the spin button control in Figure 9–1 has a location and tab order that is determined by the control-frame widget location and tab order in the Customer Information dialog box.

The control-frame widget also allows you to handle widget events that interact with other field-level widgets (like TAB and LEAVE) or that have special Progress significance (like GO and END-ERROR). For more information on handling widget events in the 4GL, see the sections on user input and user-interface triggers in the *Progress Programming Handbook*.

Control-frame COM Object

The control-frame COM object is the actual ActiveX control container. It provides the initial point of access to the ActiveX control from the 4GL. Through this COM object you can access the component handle of the ActiveX control, which in turn allows you to directly access the ActiveX control's properties and methods. For information on accessing ActiveX control properties and methods in the 4GL, see Chapter 7, "Using COM Objects In the 4GL."

Figure 9–2 shows the relationship between the ActiveX control, its control-frame, and other widgets in the Progress user interface.

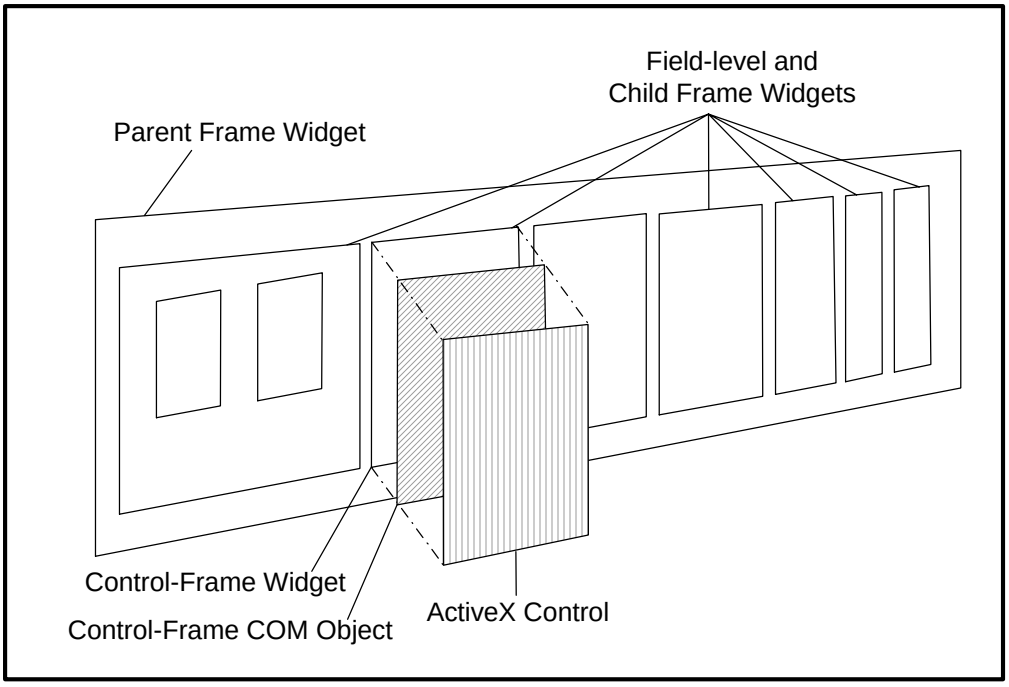


Figure 9–2: ActiveX Control Encapsulation In Progress

9.1.3 Control-frame Attributes and Properties

In addition to the control-frame widget attributes that manage an ActiveX control's relationship to the user interface, the control-frame COM object has several properties that map to corresponding control-frame widget attributes. These properties provide another way of getting to the same information and are listed in Table 9–1.

Table 9–1: Control-frame Attributes and Properties *(1 of 2)*

Widget Attribute	COM Object Property
HEIGHT-PIXELS, HEIGHT[-CHARS]	Height
NAME	Name
WIDTH-PIXELS, WIDTH[-CHARS]	Width

Table 9–1: Control-frame Attributes and Properties *(2 of 2)*

Widget Attribute	COM Object Property
X, COLUMN	Left
Y, ROW	Top

If one of these control-frame widget attributes changes, the corresponding COM object property changes, and the reverse is also true. For example, suppose CtrlFrame is the widget handle to a control-frame and chCtrlFrame is the component handle to the same control-frame. If you set CtrlFrame:X = 10, the value of chCtrlFrame:Left is set to 10. If you set chCtrlFrame:Width = 100 (100 pixels), the value of CtrlFrame:WIDTH-PIXELS is set to 100. In this case, CtrlFrame:WIDTH also changes, but the exact value (number of characters) depends on the font.

For a complete list of the attributes and properties (as well as methods and events) associated with the control-frame, see the CONTROL-FRAME Widget entry in the *Progress Language Reference*.

9.1.4 Additional Control Functionality

A Progress control container often maintains information regarding the management of the controls it contains. This information is available as an extended control which essentially provides a wrapper around the base control. Even though the extended control maintains the extended properties, methods, and events, each of these appears as though it is part of the base control. As a result, accessing any extended feature is identical to accessing a feature on the base control itself. An example of an extended property is the property Tag, as its functionality is provided by the container. The base control does not even know the property Tag exists. It is the extended control that handles any access to the Tag property.

Properties

Table 9–2 lists the available extended properties.

Table 9–2: Extended Properties

Property Name	Type	Access
HonorProKeys	logical	read/write at design time read/write at runtime
Name	string	read/write at design time read/write at runtime
Parent	COM-handle	read at runtime
Tag	string	read/write at design time read/write at runtime
Visible	logical	read/write at design time read/write at runtime

Table 9–3 defines each property and how it can be used.

Table 9–3: Property Definitions *(1 of 2)*

Property	Definition
HonorProKeys	The default value of TRUE allows the Progress code to process the **GO**, **ENDKEY**, **HELP**, or **TAB** key stroke as each is defined. Setting the value to FALSE causes the control to process the key stroke without Progress code receiving notification that the key stroke occurred.
Name	The Name property contains the name of the control. The name is important because it identifies the control. You can use the control's name to get a COM-HANDLE to the control (for example, chCSSpin=chCFSpin:CSSpin, where CSSpin is the control's name and chCFSpin is the control frame handle). The control name associates event handlers with a control.
Parent	The Parent property is the com-handle (a pointer to the IDispatch interface) of the container in which the control resides. This property is set internally by Progress.

Table 9–3: Property Definitions *(2 of 2)*

Property	Definition
Tag	The Tag property is a user property that allows the user to store an arbitrary string value and retrieve it later. Progress does not use this property internally, and it is intended to give the user a way of storing application specific information with the control. This property is initialized to an empty string.
Visible	The Visible property determines and indicates whether an ActiveX control is currently displayed. The Visible property is distinct from, but influenced by, the Visible and Hidden attributes of the Control-Frame widget. The Visible property will appear in the Property Editor and can be set at design time. It defaults to TRUE. The value set in the Property Editor determines whether the OCX is initially displayed when the program is run, but can be overriden by the value of the Control-Frame widget's Hidden attribute. Some ActiveX controls are never displayed at runtime (for example, a timer control.) For these controls, the Visible property will not appear in the Property Editor and attempts to set the property at runtime will have no effect.

9.1.5 Special Considerations For Extended Controls

The control container normally maintains relationships among controls. However, in Progress, because the control-frame can hold only a single ActiveX control, the extended properties (such as those that manage control position within the control container) can be maintained on the control-frame widget itself. Thus, the control-frame widget attributes manage the relationship between the ActiveX control and the parent frame widget (its effective container) in the user interface. For this same reason, additional extended properties that would otherwise be available on the extended control are found on the control frame widget instead.

9.1.6 ActiveX Control Restrictions

A single control-frame is capable of holding only a single ActiveX control. However, you can include many control-frames (and thus many ActiveX controls) in a Progress frame. This limitation incurs certain restrictions on the types of ActiveX controls that you can use. For instance, there is no internal support for container controls — that is, controls that allow multiple controls to reside inside of them.

9.2 Creating a Control Instance In Progress

You must use the AppBuilder to add an ActiveX control to your application. Follow these steps in the AppBuilder to add a control:

1 ♦ Open a container object, such as a Dialog or SmartWindow.

2 ♦ Choose an ActiveX control from the object Palette by clicking the OCX button to open the Choose Control dialog box, or by clicking the button for one of the three Progress-supported ActiveX controls.

3 ♦ Drop the selected control into your design window.

NOTE: These are general steps. For complete information on using the AppBuilder to add an ActiveX control, see the *Progress AppBuilder Developer's Guide*.

If you save your work at this point, the AppBuilder generates the following data and code for your application:

- The definition for a default instance of the specified ActiveX control in a binary (.wrx) file. This includes the initial definition provided by the control vendor. The binary file includes all ActiveX control instances for the corresponding .w file.
- Default 4GL code in your .w file to instantiate and orient the ActiveX control in the user interface at runtime.

The rest of this section describes how you can modify the default definition of an ActiveX control instance. For more information on the 4GL code for instantiating an ActiveX control at runtime, see the "Accessing ActiveX Controls At Runtime" section. For more information on locating and sizing the control in the user interface, see the "Orienting a Control In the User Interface At Design Time" section.

9.2.1 Understanding Design Time and Runtime Properties

To customize the definition for an ActiveX control, you must change the values of control properties in the AppBuilder at design time. Each ActiveX control supports a specific list of properties that you can set at design time (*design time properties*) and another list of properties that you can read or write at runtime (*runtime properties*). (You can set some properties at both design time and runtime.)

To modify design time properties, you generally require a license from the control vendor. The vendor typically provides this license to application developers (as opposed to application end-users) for installation with the control.

The AppBuilder provides access to all available design time properties using the OCX Property Editor window. This window contains all design time properties including the extended properties that Progress adds. This is the only way to set values for properties writable only at design time.

To modify run-time properties, you write 4GL code that sets the property value using the component handle to the control.

For more information on the available design time properties, see the list displayed in the OCX Property Editor window, and consult your ActiveX control documentation. For information on the available run-time properties, see the list displayed in the Progress COM Object Viewer (see Chapter 7, "Using COM Objects In the 4GL"), and consult your ActiveX control documentation.

9.2.2 Setting Design Time Properties

Unlike 4GL widgets, which have a standard set of defaults for the common attributes, the default settings for common ActiveX control properties vary from control to control. If you observe unexpected run-time behavior from an ActiveX control, consider whether you have set design time property values that are appropriate for your application.

NOTE: Similar considerations apply to setting run-time property values for ActiveX controls. For more information, see the "Programming ActiveX Controls In the AppBuilder" section.

9.2.3 Setting the ActiveX Control Name

The AppBuilder creates a default name (OCX name) for the ActiveX control instance when you first insert it into the design window. (This is the name defined by the control vendor.) You can change this name in the AppBuilder through the design time Name property. The AppBuilder identifies the control instance by both the control-frame name and the OCX name, rather than the OCX name alone. Because the control-frame name is always unique, the control-frame/OCX name pair is always unique.

If you do decide to change the ActiveX control name, be aware that Progress uses this name to identify event handlers for the control. If you define event handlers in external procedures other than the one where you initially define the control instance, the AppBuilder cannot update them to conform with the new name. You must modify the names of these external event handlers manually to conform with the new name. For more information on defining event handlers for ActiveX controls, see the "Handling Events" section.

NOTE: Progress does not recommend changing the control name at run time. If you do, the event handlers for the control will not work, since an event handler name is formed in part from the name of the control-frame and the control.

9.2.4 Setting the Control-frame Name

When you first insert an ActiveX control into the design window, the AppBuilder creates a unique default Object name for the control-frame that contains the ActiveX control. As with any 4GL widget, you can change this Object name in the AppBuilder. However, you must use extra care when changing this name. The AppBuilder uses the control-frame Object name for three program elements:

- As the value for the control-frame widget NAME attribute (and COM object Name property). When loading the associated ActiveX control at runtime, the AppBuilder locates the control instance in the section of the .wrx file identified by this name. When responding to ActiveX control events, Progress uses this name to identify event handlers for the control.
- As the variable name for the widget handle of the control-frame widget.
- As a part of the variable name (prefixed by “ch”) for the component handle of the control-frame COM object.

When you change the control-frame Object name manually in the AppBuilder, the AppBuilder automatically updates these three program elements and all AppBuilder-generated code. When you save the application file, aside from saving the changes to these program elements, it also updates the corresponding section name in the .wrx file.

However when you change this Object name, the AppBuilder does not update any custom code you have added using the Section Editor. It also does not update the identity of any event handlers that you might have defined for the control in external procedures other than the one where you initially defined the control. You must update this code manually in the Section Editor or external procedure file. (For more information on how Progress identifies event handlers for ActiveX controls, see the “Handling Events” section.)

NOTE: Progress does not recommend changing the control-frame name at run time. If you do, the event handlers for the control will not work, since an event handler name is formed in part from the name of the control-frame and the control.

9.3 Orienting a Control In the User Interface At Design Time

The control-frame manages the location and dimensions of the associated ActiveX control in the user interface. Thus, to orient the ActiveX control at design time, you must set certain control-frame attributes. You can set these attributes directly, using the AppBuilder Property Sheet. You can also set them indirectly by visually manipulating the control in the design window.

NOTE: For Progress widgets, you can open the AppBuilder Property Sheet by double-clicking on the widget. However for ActiveX controls, this opens the OCX Property Editor window. To access the control-frame widgets Property Sheet, you must select the ActiveX control, then click the Object properties button or choose Property Sheet from the Tools menu.

9.3.1 Setting Control Location

The AppBuilder automatically updates the control location wherever you insert or move it in the design window. This movement updates the COLUMN, ROW, X, and Y attributes of the control-frame widget (and the corresponding control-frame COM object properties). You can also modify these attributes manually in the Property Sheet of the control-frame.

9.3.2 Setting Control Height and Width

The AppBuilder automatically updates the size of the control whenever you use the mouse to adjust the resize handles. These resize handles actually belong to the control-frame and the ActiveX control generally conforms to the dimensions you set for the control-frame.

There are some exceptions to this. Some controls depend on a specific size. For example, if the control is represented by an icon, it expects to be sized appropriately (for example, 32 pixels square). Another example is a combo box that expects its size to be only high enough to hold its text. For controls like these, if the user tries to resize the control-frame in a way that violates its internal sizing rules, the control automatically snaps back to the size that it requires. In this case, the control-frame resizes to fit the control rather than the other way around.

For resizable controls, these adjustments update the HEIGHT, HEIGHT-PIXELS, WIDTH, and WIDTH-PIXELS attributes of the control-frame widget (and the corresponding control-frame COM object properties). You can also modify these attributes manually in the Property Sheet of the control-frame.

9.3.3 Setting Control Tab Order

Control tab order is established by the attributes of the control-frame widget. The control-frame is a dynamic widget parented (by the AppBuilder) to a static frame. As such, each control-frame in the frame assumes a default tab order that begins after any static input widgets (such as buttons) parented by the frame. However, the AppBuilder generates code that adjusts the tab order of widgets to match the order that you insert them in the design window. At design time, you can reset the tab order of widgets different from the insertion order using the Tab Editor dialog box in the AppBuilder. For information on managing the tab order of ActiveX controls at runtime, see the "Managing ActiveX Controls At Runtime" section.

NOTE: As you tab between ActiveX controls and field-level widgets at design time or at run time, Progress shows no indication of focus in the ActiveX controls. Otherwise, ActiveX controls behave like any other widget in a Progress frame.

9.3.4 Defining Invisible Controls

For controls with a run-time user interface (such as the CSSpin control provided with Progress), you can make them initially invisible by setting the Hidden option in the control-frame Property Sheet. (This sets the control-frame widget HIDDEN attribute to TRUE.)

For controls without a run-time user interface (such as the PSTimer control provided with Progress), the AppBuilder automatically sets the control-frame HIDDEN attribute to TRUE. For such controls, you cannot change this attribute from the Property Sheet.

Also, for a control without a run-time user interface, location, size, and tab order generally have no run-time significance. At design time, however, you might want to position (and resize, if possible) the control's design time representation for ease of maintenance.

NOTE: Progress automatically removes some controls from the tab order if they have a flag indicating that they should not receive input focus. These controls might be visible. For example, the Status bar control is visible, but never gets focus.

9.4 Accessing ActiveX Controls At Runtime

To access an ActiveX control at runtime, your application must instantiate the control and get the component handle to the ActiveX control.

9.4.1 Instantiating the Control

At runtime, your application uses AppBuilder-generated code to instantiate any ActiveX controls. This code (shown in following sections) uses the CREATE Widget statement to realize a separate control-frame widget for each ActiveX control, initializing each control-frame with the name specified at design time. It then invokes the LoadControls( ) method on each control-frame COM object to instantiate the corresponding ActiveX control, loading all design time definitions from the .wrx file.

Figure 9–3 describes the instantiation process for a single ActiveX control.

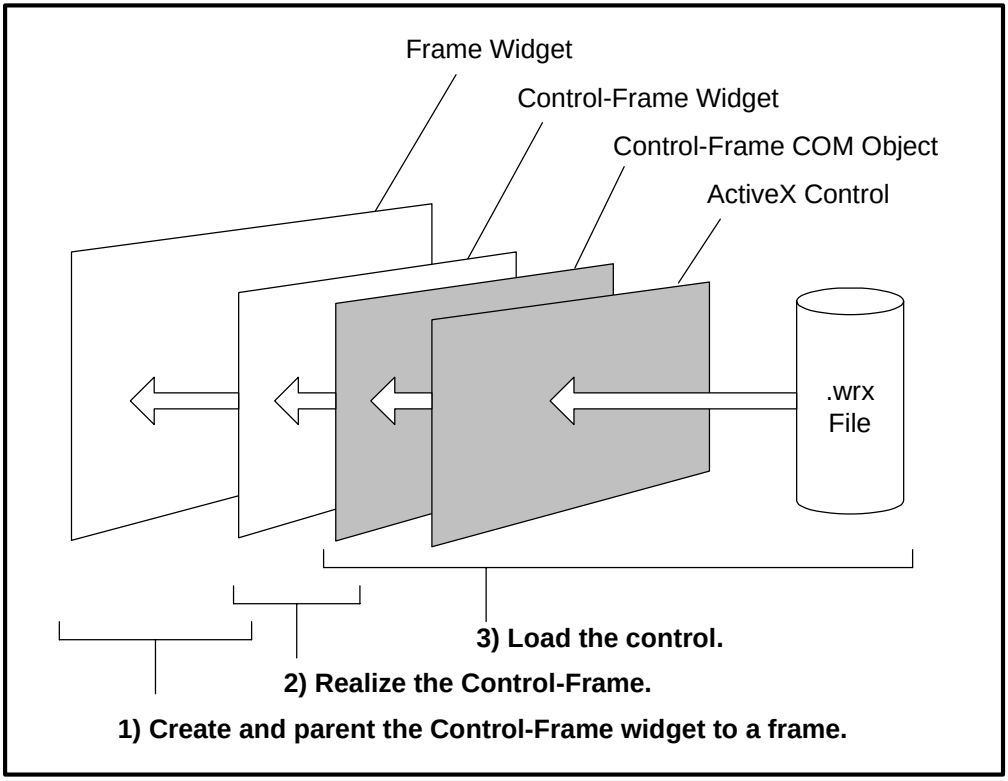


Figure 9–3: Instantiating an ActiveX Control At Runtime

Creating the control-frame widget does not, by itself, realize the control-frame. Control-frame realization occurs only after (1) it is parented to a frame widget and (2) its NAME attribute is set or its COM-HANDLE attribute is referenced. The NAME attribute is generally set first and causes realization by creating the control-frame COM object. Only then can your application use the LoadControls( ) method to (3) load the control into the control-frame COM object (the control container).

NOTE: The control-frame name must match the name specified at design time to allow LoadControls( ) to locate the specified ActiveX control in the .wrx file.

9.4.2 Accessing the Control Handle

Once the ActiveX control is instantiated, the control-frame affords access to one widget and three COM objects, including the:

- Control-frame widget
- Control container (the control-frame COM object)
- Control collection (a standard COM object)
- ActiveX control (the target COM object)

Figure 9–4 shows the chain of handle references that connect a control-frame widget (with widget handle CF) to an ActiveX control (with the Name "CSSpin").

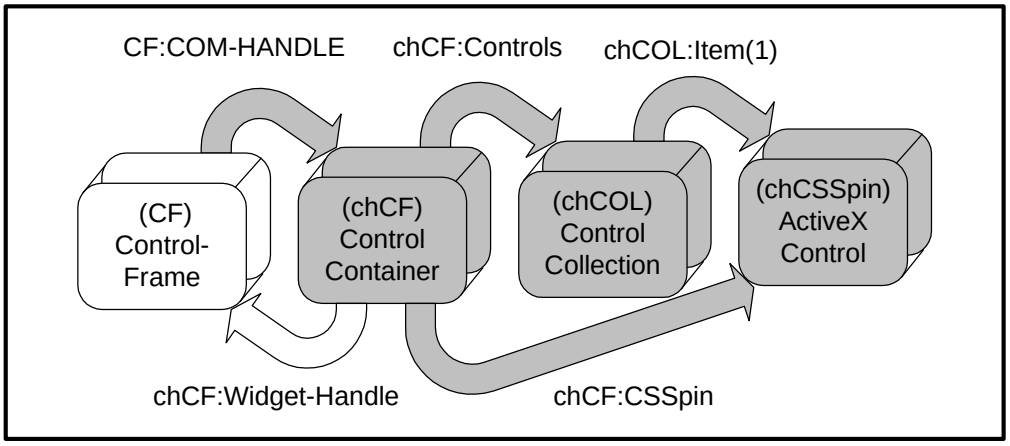


Figure 9–4: Handle References To Access ActiveX Controls

The chCF reference is a component handle to the control-frame COM object (control container) and the chCOL reference is a component handle to the control collection. Thus in the 4GL, you can instantiate and get the handle to an ActiveX control in two ways:

- Using the name of the ActiveX control ("CSSpin") as if it were a property (*Control-Name* property) of the control container (chCF:CSSpin)
- Using the indexed Item property of the control collection (chCOL:Item(1))

Once you have the component handle of the ActiveX control, you can access all of its properties and methods. (The chCSSpin reference is a component handle variable you might set from chCF:CSSpin or chCOL:Item(1).) The following examples reference a reduction of typical code generated by the AppBuilder. For a closer look at actual code generated by the AppBuilder, see the "Programming ActiveX Controls In the AppBuilder" section.

Using the Control-Name Property Of the Control Container

This example show how you might access the component handle to the CSSpin control using the *Control-Name* property:

```
DEFINE VARIABLE chCSSpin AS COM-HANDLE.     /* ActiveX Control */

DEFINE VARIABLE CtrlFrame AS WIDGET-HANDLE. /* Control-Frame Widget */
DEFINE VARIABLE chCtrlFrame AS COM-HANDLE.  /* Control-Frame COM Object */
DEFINE VARIABLE OCXFile AS CHARACTER        /* .wrx file pathname */

/* ... Define a FRAME widget named 'Foo' ... */

/* Create OCX Container */

CREATE CONTROL-FRAME CtrlFrame ASSIGN  /
       FRAME = FRAME Foo:HANDLE.
      CtrlFrame:NAME = "CtrlFrame":U.

/* Load (instantiate) ActiveX control */

OCXFile = SEARCH ( "csspinapp.wrx":U ).

chCtrlFrame = CtrlFrame:COM-HANDLE.
chCtrlFrame:LoadControls( OCXFile, "CtrlFrame":U ).

chCSSpin = chCtrlFrame:CSSpin /* Control handle via the Control Name */
```

The bolded code is the code you might add to appropriate sections of your application; the rest is a simplified version of code the AppBuilder might generate. Note how the AppBuilder uses the control-frame name (CtrlFrame) to generate handle variable names and to locate the control instance in the `csspinapp.wrx` file. The actual control name (CSSpin) is the OCX name specified for the control at design time (in this case, the CSSpin default).

Using the Item Property Of the Control Collection

This similar example uses the control collection to return the control's component handle:

```
DEFINE VARIABLE chCSSpin AS COM-HANDLE.     /* ActiveX Control */

/* ... The AppBuilder-generated code from the previous example ... */

chCSSpin = chCtrlFrame:Controls:Item(1). /* Control handle via Item( ) */
```

The Handle Connections

The previous examples rely on attributes, properties, and methods supported by Progress to get a component handle from a control-frame widget handle (see Figure 9–4). Thus, to provide access to the control-frame widget and its related COM objects:

- The control-frame widget has a COM-HANDLE attribute that returns the component handle of the control-frame COM object.
- The control-frame COM object has a Controls property that returns the component handle of the control collection. You can then use the Item property on the collection component handle to get the component handle of the control.

  NOTE: The control collection is a standard COM object that is used to access sets of like objects (objects in the same class).

- The control-frame COM object has a special property named after the ActiveX control that returns the component handle of the control.
- The control-frame COM object provides a Widget-Handle property that returns the widget handle of the instantiating control-frame widget.

9.5 Managing ActiveX Controls At Runtime

At runtime, the control-frame itself is largely irrelevant from the end-user viewpoint. It is the ActiveX control itself that is visible and useful. In general, the control exhibits its standard behavior as if the control-frame were not there.

The control does not follow any special Progress rules except those described in the following sections. For example, edit controls do not have any of the customized Progress text editing behavior. This means that they do not abide by Progress formats or dictionary validation rules. They are not affected by any Progress startup parameters including, for example, -d (Date Format), -E (European Numeric Format), and -yy (Century). You can make any format, validation, or international settings using either the control's own properties or by changing regional settings in the system control panel. You can also write 4GL code to do formatting and validation for a control.

However, Progress does enforce some rules and provides 4GL mechanisms that allow ActiveX controls to live more comfortably with other 4GL widgets in an application.

9.5.1 Managing Tab Order and Z Order

Like any other Progress widget, ActiveX controls participate in the tab order of the Progress frame to which they are parented. The tab order of a control is managed by the control-frame widget. You can modify the tab order of a control using the control-frame's MOVE-AFTER-TAB-ITEM( ) and MOVE-BEFORE-TAB-ITEM( ) widget methods.

The same is true of Z order (the overlay order of controls). You can modify the Z order of a control using the control-frame's MOVE-TO-BOTTOM( ) and MOVE-TO-TOP( ) widget methods.

9.5.2 Working With Progress Key Functions

Setting the HonorProKeys property to TRUE allows the user to specify that Progress should handle certain key functions.

TAB, BACK-TAB, GO, HELP, and END-ERROR

Progress enforces the TAB and BACK-TAB key functions in all ActiveX controls. If a control defines a meaning for the **TAB** or **SHIFT-TAB** key other than normal tabbing, the control-defined function will not work. (Few, if any, controls redefine the **TAB** or **SHIFT-TAB** key in this way.)

Progress also enforces behavior on the three keys that implement the GO, HELP and END-ERROR functions (normally the **F2**, **F1** and **ESC** keys, respectively). Again, if the control normally uses these keys for another purpose, that functionality is lost in Progress. Note that the standard use of the **F1** and **ESC** keys in Windows applications matches the standard use of HELP and END-ERROR in Progress, and ActiveX controls do not generally use **F2**.

All three of these key functions work if focus is in a subwindow of an ActiveX control. For example, suppose a calendar control has two subwindows where one holds the month and the other holds the year. If you click into one of these subwindows and press **TAB**, focus moves to the next control or widget in the frame.

To allow the control to process the **GO**, **HELP**, **TAB**, and **ENDKEY** keys, set the HonorProKeys extended property to FALSE.

RETURN and Default Buttons

In Windows, the RETURN key function (normally the **ENTER** key) has special significance in that it can invoke the default button in a dialog box. You can program this functionality in Progress by designating a button as the DEFAULT-BUTTON of a frame. However, because it is common for an ActiveX control to specify its own use for the **ENTER** key, Progress does not trap this key in ActiveX controls. Therefore, RETURN does not activate the default button in a Progress frame if an ActiveX control has focus.

To override this behavior, set the HonorReturnKeys property to TRUE, so Progress can handle the **RETURN** key.

9.5.3 Setting Graphical Properties Of an ActiveX Control

In Progress you can set graphical properties of an ActiveX control with the LOAD-PICTURE statement. The LOAD-PICTURE statement takes a filename of a graphical object and returns a COM-HANDLE to an OlePictureObject. For example, the Microsoft Image control has a picture property that controls the image displayed in the control. The type of this property is equivalent to a Progress COM-HANDLE.

9.5.4 Releasing Control Resources

The following material describes the process of creating and freeing control-frame widgets. This is automatically done by the AppBuilder, but, like any other dynamic widget, you can delete a control-frame some time after you create it.

You can delete a control-frame using two techniques:

- Associate the control-frame with a widget pool in the CREATE statement. When you delete the widget pool, Progress deletes the control-frame widget and also releases its control-frame COM object. This is the default technique used by the AppBuilder. (The AppBuilder uses the default unnamed widget pool that is deleted when the .w ends.)

- Explicitly delete the control-frame using the DELETE WIDGET statement. This statement deletes the control-frame widget and also releases the control-frame COM object.

Releasing ActiveX Controls

When you delete a control-frame widget, Progress also automatically releases the control-frame COM object as well as any references to the ActiveX control held by the control-frame. You must release all other COM objects using the RELEASE OBJECT statement:

```
DEFINE VARIABLE CtrlFrame AS WIDGET-HANDLE.
DEFINE VARIABLE chCtrlFrame AS COM-HANDLE.
DEFINE VARIABLE chCSSpin AS COM-HANDLE.
DEFINE VARIABLE chCollection AS COM-HANDLE.

/* Create frame Foo and instantiate control ... */

CREATE CONTROL-FRAME CtrlFrame
  ASSIGN
    FRAME = FRAME Foo:HANDLE
    NAME = "CtrlFrame":U.

chCtrlFrame = CtrlFrame:COM-HANDLE.
chCtrlFrame:LoadControls("csspinapp.wrx":U, "CtrlFrame":U).

chCollection = chCtrlFrame:controls.
chCSSpin = chCollection:Item(1).
chCSSpin:ShadeColor = RGB-VALUE(0,128,0).
RELEASE OBJECT chCollection.

/* do some more stuff ... WAIT-FOR ... */

DELETE WIDGET CtrlFrame.
```

This example releases the control collection after it is no longer needed. It also deletes the control-frame using the DELETE WIDGET statement, which also releases the chCtrlFrame COM object as well as the ActiveX control itself (chCSSpin).

Releasing COM Objects Individually

If you try to release an ActiveX control (using the RELEASE OBJECT statement) before the control-frame is deleted, this works but is unnecessary. If you try to access a control after the control-frame is deleted, Progress displays an error message that you are trying to reference an invalid component handle.

Because you have a component handle to a control-frame COM object, you might think you can release it using the RELEASE OBJECT statement. However for control-frame COM objects, Progress does not allow this because of the link between the control-frame widget and COM object.

Thus, you can only release the control-frame COM object by deleting the control-frame widget. If you do try to release the component handle of a control-frame, Progress returns an error indicating that you should delete the object through the widget handle instead.

In general, if you do not delete or release any COM objects in an application, all active COM objects remain instantiated until the end of the Progress session, at which time Progress automatically releases them.

For more information on releasing COM object resources, see Chapter 7, "Using COM Objects In the 4GL."

9.6 Handling Events

Progress processes two types of events for an ActiveX control:

- **Field-level widget events on the control-frame** — A subset of standard Progress events.
- **ActiveX control events** — A set of events that are unique to each ActiveX control and that often pass parameters.

The requirements for these two types of events are different. Thus, Progress uses separate mechanisms to handle events on the control-frame as opposed to events received from the associated ActiveX control. In addition, the two types of events are mutually exclusive and any action on the control generates either one type of event or the other.

9.6.1 Handling Control-frame Widget Events

You can handle field-level widget events for the control-frame like Progress events for any other dynamic 4GL widget, with Progress triggers built from the ON statement or the *trigger-phrase* of the CREATE statement.

The control-frame supports a subset of field-level widget events that Progress requires for:

- **Managing application execution** — Universal key function events: END-ERROR (which occurs when the END key is pressed), GO, and HELP
- **Tabbing between the control and other field-level widgets** — Navigation key function events: TAB and BACK-TAB
- **Detecting any change of focus** — High-level widget events: ENTRY and LEAVE
- **Handling programmer-defined events** — Developer events: U1 through U10

For more information on handling Progress events with triggers, see the *Progress Programming Handbook*.

NOTE: Unlike earlier support for VBX controls, you do not have to handle key function events as KeyPress events. With VBX controls, you had to trap key functions as KeyPress events in a VBX control event handler, then apply them to the VBX control-container widget to handle with corresponding key function event triggers. With ActiveX controls, your triggers on the control-frame widget now handle the supported function events automatically.

9.6.2 Handling ActiveX Control Events

Unlike Progress widget events, ActiveX control events often pass parameters. Also, while there are a few standard events common to most ActiveX controls, there are an infinite variety of possible events that are unique to each ActiveX control. Because Progress cannot have direct knowledge of all these possible events and must be able to handle the parameters of many of them, the standard Progress trigger mechanism cannot handle them.

Instead, Progress allows you to handle ActiveX control events using OCX event procedures. An *OCX event procedure* is a standard Progress internal procedure that serves as an event handler for ActiveX controls. The parameter-passing mechanism provided for Progress procedures handles most parameters that ActiveX control events typically pass. Progress identifies an OCX event procedure from the way its name is put together. This is the only syntactic feature that distinguishes the Progress internal procedure as an OCX event procedure.

Creating Event Procedures In the AppBuilder

In the AppBuilder, you define OCX event procedures as OCX event triggers. You can identify each ActiveX control event in the AppBuilder event list by the “OCX.” prefix attached to the event name. When you select an event name to define an OCX event trigger, the AppBuilder automatically generates an internal procedure block for the event procedure, including any procedure parameter definitions. (For more information, see the *Progress AppBuilder Developer’s Guide*.)

Coding Event Procedures

When you create an OCX event trigger in the AppBuilder, the AppBuilder defines an internal procedure template with the following components:

- A procedure name that identifies the ActiveX control and event being handled
- Any required procedure parameter definitions with default mode and data type mappings

If you code OCX event procedures in external procedure files other than the one where you instantiate the ActiveX control, you must code the complete event procedure yourself. In addition, you must also add the procedure to the list that Progress searches to find event procedures to handle events. For more information, see the "Managing External Procedure Files" section.

Coding Event Procedure Names

Progress supports two types of OCX event procedures, distinguished by the procedure name:

- Control-bound event procedures that handle a specific event for a specific control instance
- Generic event procedures that handle a specific event for all ActiveX controls in the application

NOTE: Names of event procedures are not case sensitive.

The names for control-bound event procedures contain three parts, delimited by a period (.). Each part can be quoted if it contains embedded spaces:

1. The name of the control-frame (the NAME attribute value of the control-frame widget)
2. The name of the ActiveX control (the Name property value set in the Property Editor)
3. The name of the event that is handled by the procedure.

As with all internal procedures that you add in the AppBuilder, the PROCEDURE statement and procedure name is not visible in the Section Editor. However, this is one that you or the AppBuilder might code:

```
PROCEDURE CtrlFrame.CSSpin.SpinUp .
```

This begins the definition for a procedure to handle the SpinUp event for the control named CSSpin in the control-frame named CtrlFrame.

The names for generic event procedures contain two parts, delimited by a period (.):

1. ANYWHERE

2. The name of the event that is handled by the procedure.

The AppBuilder does not provide a mechanism to generate generic event procedures. You must code these yourself in the AppBuilder using the New Procedure dialog box, or you can add them to an external procedure file. Here is an example:

```
PROCEDURE ANYWHERE.SpinUp .
```

A generic event procedure with this name handles the SpinUp event for any control instance that does not have a control-bound event procedure defined for the SpinUp event.

Coding Event Parameter Definitions

As for any procedure definition, you code event procedure parameters using the DEFINE PARAMETER statement. When you create an event procedure as an OCX trigger in the AppBuilder, the AppBuilder provides default definitions for all required parameters. These default definitions attempt to specify a Progress data type that is compatible with the COM data type of the corresponding OCX event parameter. This data type generally follows the data type mappings shown in Table 9–4.

Table 9–4: Data Type Mappings For OCX Event Parameters *(1 of 2)*

COM Data Type	Progress Data Type
Boolean	LOGICAL
Byte	INTEGER
Currency	DECIMAL
Date	DATE
Double	DECIMAL
Integer (2-byte integer)	INTEGER
Long (4-byte integer)	INTEGER
Object (COM)	COM-HANDLE

Table 9–4: Data Type Mappings For OCX Event Parameters *(2 of 2)*

COM Data Type	Progress Data Type
Single (Float)	DECIMAL
String	CHARACTER
Unsigned Byte	INTEGER
Unsigned Long (4-byte integer)	INTEGER
Unsigned Short (2-byte integer)	INTEGER
Variant	<ANYTYPE>[1]
VT-ARRAY (if single-dimensional array of bytes)	RAW

[1] You must replace <ANYTYPE>, provided by the AppBuilder in some OCX event trigger templates, with a valid Progress data type.

For event procedures you write yourself, you can also find the suggested Progress data type for each parameter displayed in the Progress COM Object Viewer for each selected control event. For more information on this viewer, see Chapter 7, "Using COM Objects In the 4GL."

In the AppBuilder, Progress uses data type mappings from the COM data type specified in the Type Library for the ActiveX control. If Progress cannot determine a data type mapping, the AppBuilder specifies <ANYTYPE> as a place holder. You must change <ANYTYPE> to the data type that most closely matches the expected value. Progress does its best to convert the COM data type to the Progress data type you specify. For more information on a COM data type, see the available documentation on the Microsoft Component Object Model and the event parameter you want to convert, then see Appendix B, "COM Object Data Type Mapping." For information on Type Libraries and Progress, see Chapter 7, "Using COM Objects In the 4GL."

The AppBuilder also provides a default mode option (INPUT, OUTPUT, or INPUT-OUTPUT) for each parameter definition. Here, although Progress does its best to interpret the information in the Type Library for the ActiveX control, this is not a direct mapping. You might have to modify the mode option. If the option chosen by Progress is incorrect, this generates a run-time error. If the event parameter is passed as read-only, define it as an INPUT parameter. If you can change its initial value and must pass it back to the ActiveX control, define it as an INPUT-OUTPUT parameter. If only the value you return to the ActiveX control has any meaning, define it as an OUTPUT parameter.

CAUTION: If an INPUT parameter passes in a COM-HANDLE value, use the RELEASE OBJECT statement to release the specified COM object when you no longer need it (usually before the event procedure returns). Otherwise, the component handle might become unavailable and the COM object might never be released. Also, not releasing the COM object in the event procedure can cause unpredictable behavior with some controls.

CAUTION: If an event procedure has any INPUT-OUTPUT or OUTPUT parameters and contains any I/O-blocking statements, such as SET or UPDATE, the output parameters are **not** returned to the control.

Using System Handles In OCX Event Procedures

Progress allows you to obtain the identities of both the ActiveX control that generates an event and the associated control-frame widget from within an OCX event procedure. The COM-SELF system handle returns the component handle of the ActiveX control and the SELF system handle returns the widget handle of the control-frame. This is especially useful in generic event procedures where the source of an event cannot be known.

Unlike for 4GL ON triggers, the RETURN NO-APPLY statement has no affect in an OCX event procedure. Some COM object events provide a similar mechanism using output parameters. For example, the standard KeyPress event passes a key code as an INPUT-OUTPUT parameter. If you set this parameter to 0 in the event handler, the key is discarded.

NOTE: To trigger a Progress widget event from the 4GL, you use the APPLY statement. However, to trigger an ActiveX control event in the 4GL, you execute its OCX event procedure as a standard internal procedure, passing any required parameters.

Event Handler Example

The following example shows an OCX event trigger for the NewItem event of the CSComboBox control MyCombo in control-frame CtrlFrame-2. This event occurs when a user enters an item in the text box that is not in the combo box list:

```
PROCEDURE CtrlFrame-2.MyCombo.NewItem .

DEFINE INPUT-OUTPUT PARAMETER p-ComboText AS CHARACTER NO-UNDO.
DEFINE INPUT-OUTPUT PARAMETER p-KeepFocus AS INTEGER NO-UNDO.

    COM-SELF:AddItem(p-ComboText, 1).
    p-KeepFocus = -1.

END PROCEDURE.
```

The programmer uses the COM-SELF handle to add the new text item to the list with the AddItem( ) method. Setting p-KeepFocus to -1 ensures that the control maintains input focus after the item is added to the list.

If this event procedure is in the AppBuilder-generated application file, the programmer might otherwise choose to call the AddItem( ) method indirectly using the component handles provided by the AppBuilder (chCtrlFrame-2) and the control-frame COM object (MyCombo property):

```
PROCEDURE CtrlFrame-2.MyCombo.NewItem .

DEFINE INPUT-OUTPUT PARAMETER p-ComboText AS CHARACTER NO-UNDO.
DEFINE INPUT-OUTPUT PARAMETER p-KeepFocus AS INTEGER NO-UNDO.

    chCtrlFrame-2:MyCombo:AddItem(p-ComboText, 1).
    p-KeepFocus = -1.

END PROCEDURE.
```

9.6.3 Managing External Procedure Files

As described earlier (see the "Handling ActiveX Control Events" section), you can define OCX event procedures in external procedure files other than the one where you instantiate the ActiveX control. If you do create these external procedure files, you need to tell Progress how to locate the event procedures when responding to ActiveX control events. You do this by creating a search list of your external procedures.

The control-frame widget provides two methods for you to manage this search list:

- ADD-EVENTS-PROCEDURE(*procedure-handle*)
- REMOVE-EVENTS-PROCEDURE(*procedure-handle*)

These methods require that your external procedures are running persistently or on the procedure call stack. To have Progress locate an event procedure in one of these external procedures, you invoke the ADD-EVENTS-PROCEDURE( ) on the control-frame widget, passing the procedure handle of the external procedure. To remove an external procedure from the search list, you pass its procedure handle to the REMOVE-EVENTS-PROCEDURE( ) method on the control-frame widget.

When Progress responds to a control event, it searches the assembled list of external procedures to find a matching control-bound event handler, searching in order of the most recently added procedure first. Finally, it searches the application file where the control is instantiated. If it cannot find a control-bound event handler, it searches the list again for a matching generic event handler, always searching the main application file last. In this way, Progress selects the first matching event handler to handle the incoming event.

Using these methods, you can dynamically add and remove procedures from the search list, overriding or replacing previously added procedures. Thus, modifying this search list can have a powerful effect on the behavior of an application.

9.7 Programming ActiveX Controls In the AppBuilder

The following code fragments illustrate selected sections of the sample ActiveX control application, `e-ocx1.w`, shown running in Figure 9–1. Recall that this application uses a spin button control to scan Customer records in the sports database.

9.7.1 Creating Data Definitions For an ActiveX Control

These fragments provide data and code definitions, including those for the spin button control, referenced by other sections of the application. The first fragment contains variable definitions that are hand-coded in the Definitions section of the AppBuilder Section Editor:

```
/* Local Variable Definitions ---                                       */

DEFINE VARIABLE result          AS LOGICAL.
DEFINE VARIABLE max-records     AS INTEGER.
DEFINE VARIABLE chCSSpin        AS COM-HANDLE.
```

Note the variable to hold the component handle CSSpin ActiveX control. All other necessary widget and component handles for this application are provided by the AppBuilder.

This is the query definition assembled by the AppBuilder from user input. The PRESELECT option allows the application to know the number of records it is scanning with the spin button at startup:

```
/* ********************  Preprocessor Definitions  ******************** */
            .
            .
            .

&Scoped-define OPEN-QUERY-Dialog-Frame OPEN QUERY Dialog-Frame PRESELECT EACH
sports.Customer NO-LOCK.
```

The following code section shows relevant widget handle, component handle, and query definitions generated by the AppBuilder from user input. The AppBuilder assembles the widget definitions as you add objects to the design window. Thus, the AppBuilder assembles the definitions for the control-frame handles (custSpin and chcustSpin) after you insert the spin button control into the design window. (The programmer has changed the name of the control-frame from CtrlFrame to custSpin.) The application also uses the Record_Count variable to display the ordinal record number of each Customer record scanned by the spin button control:

NOTE: You can begin setting ActiveX control properties in the OCX Property Editor Window any time after you add the control into the design window.

```
/* ***********************  Control Definitions  ********************** */
             .
             .
             .
/* Definitions of handles for OCX Containers                            */
DEFINE VARIABLE custSpin AS WIDGET-HANDLE NO-UNDO.
DEFINE VARIABLE chcustSpin AS COMPONENT-HANDLE NO-UNDO.
             .
             .
             .

DEFINE VARIABLE Record_Count AS INTEGER FORMAT "9999":U INITIAL 0
     LABEL "Record Number"
      VIEW-AS TEXT
     SIZE 5.8 BY .62 NO-UNDO.
             .
             .
             .
/* Query definitions                                                    */
             .
             .
             .
DEFINE QUERY Dialog-Frame FOR
      sports.Customer SCROLLING.
```

NOTE: You cannot view or change this AppBuilder-generated code from the Section Editor. However, it is part of the .w file generated for the application, and you can use Code Preview option of the Tools menu to view this code.

The AppBuilder automatically generates the query definition from the query criteria entered earlier.

This fragment shows the actual control-frame definition created by the AppBuilder using the custSpin widget handle defined earlier. The AppBuilder creates and maintains this definition as you insert, position, and size the spin button control in the design window. Note the setting of the control-frame NAME attribute after the widget is created and parented to the Dialog-Frame frame. The AppBuilder also sets the HIDDEN attribute to no (FALSE) because the spin button control is, by default, a visible control:

```
/* **********************  Create OCX Containers  ********************** */

&ANALYZE-SUSPEND _CREATE-DYNAMIC

&IF "{&OPSYS}" = "WIN32":U AND "{&WINDOW-SYSTEM}" NE "TTY":U &THEN

CREATE CONTROL-FRAME custSpin ASSIGN
       FRAME         = FRAME Dialog-Frame:HANDLE
       ROW           = 11.19
       COLUMN        = 15.4
       HEIGHT        = 2.33
       WIDTH         = 61.2
       HIDDEN        = no
       SENSITIVE     = yes.
      custSpin:NAME = "custSpin":U .
/* custSpin OCXINFO:CREATE-CONTROL from:
{EAF26C8F-9586-101B-9306-0020AF234C9D} type: CSSpin */
      custSpin:MOVE-AFTER(Btn_Update:HANDLE IN FRAME Dialog-Frame).

&ENDIF

&ANALYZE-RESUME /* End of _CREATE-DYNAMIC */
```

NOTE: You cannot view or change this AppBuilder-generated code from the Section Editor. However, it is part of the .w file generated for the application, and you can use Code Preview option of the Tools menu to view this code.

The AppBuilder also ensures, with appropriate preprocessor settings, that Progress only compiles and executes OCX-related sections of this procedure file if it is running in the graphical mode (not "TTY") of a Windows system ("WIN32").

9.7.2 Instantiating the Control

The AppBuilder generates the code to instantiate the ActiveX control starting with a call to the enable_UI procedure from the Main Block:

```
/* ***************************  Main Block  *************************** */

/* Parent the dialog-box to the ACTIVE-WINDOW, if there is no parent.   */
IF VALID-HANDLE(ACTIVE-WINDOW) AND FRAME {&FRAME-NAME}:PARENT eq ?
THEN FRAME {&FRAME-NAME}:PARENT = ACTIVE-WINDOW.

/* Now enable the interface and wait for the exit condition.            */
/* (NOTE: handle ERROR and END-KEY so cleanup code will always fire.    */
MAIN-BLOCK:
DO ON ERROR   UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK
   ON END-KEY UNDO MAIN-BLOCK, LEAVE MAIN-BLOCK:
    RUN enable_UI.

    /* Set max-records from open query in enable_UI. */

    max-records = NUM-RESULTS("Dialog-Frame").

    WAIT-FOR GO OF FRAME {&FRAME-NAME}.
END.
RUN disable_UI.
```

The programmer has added the comments and code that immediately follow the call to enable_UI. Here, they must set max-records to the number of records in the query only after the query is opened (again, in enable_UI), but before the spin button control is available to scan the result. The reason for this becomes clearer in later code fragments.

The following reduction of the enable_UI procedure shows that the AppBuilder generates the code to load the ActiveX control (RUN control_load) before its static parent frame and family of 4GL widgets are displayed and enabled. The initial value of Record_Count is also set at control load time (see the "Initializing the Control" section. Note also that the application query is opened to obtain the data for the frame:

```
/* **********************  Internal Procedures  *********************** */
             .
             .
             .
PROCEDURE enable_UI :
/*------------------------------------------------------------------------
  Purpose:     ENABLE the User Interface
  Parameters:  <none>
  Notes:       Here we display/view/enable the widgets in the
               user-interface.  In addition, OPEN all queries
               associated with each FRAME and BROWSE.
               These statements here are based on the "Other
               Settings" section of the widget Property Sheets.
------------------------------------------------------------------------*/
  RUN control_load.

  {&OPEN-QUERY-Dialog-Frame}
  GET FIRST Dialog-Frame.
  DISPLAY Record_Count
      WITH FRAME Dialog-Frame.
             .
             .
             .
END PROCEDURE.
```

NOTE: You can view this code in the Procedures section of the AppBuilder Section Editor. However, unlike the Main Block, this is protected AppBuilder-generated code that you cannot change.

The control_load procedure, called from enable_UI, is an AppBuilder-generated procedure that loads the ActiveX control into the control-frame by executing the LoadControls( ) method of the control-frame COM object. Note the call to initialize-controls, an optional internal procedure that you can define (and which is defined in this example) to modify control properties before the control becomes visible:

```
/* **********************  Internal Procedures  ********************** */
            .
            .
            .
PROCEDURE control_load :
/*------------------------------------------------------------------------
  Purpose:     Load the OCXs
  Parameters:  <none>
  Notes:       Here we load, initialize and make visible the
               OCXs in the interface.
------------------------------------------------------------------------*/
&IF "{&OPSYS}" = "WIN32":U AND "{&WINDOW-SYSTEM}" NE "TTY":U &THEN
DEFINE VARIABLE UIB_S    AS LOGICAL    NO-UNDO.
DEFINE VARIABLE OCXFile  AS CHARACTER  NO-UNDO.

OCXFile = SEARCH( "e-ocx1.wrx":U ).

IF OCXFile <> ? THEN DO:
  ASSIGN
    chcustSpin = custSpin:COM-HANDLE
    UIB_S = chcustSpin:LoadControls( OCXFile, "custSpin":U).
  RUN initialize-controls IN THIS-PROCEDURE NO-ERROR.
END.
ELSE MESSAGE "The file, e-ocx1.wrx, could not be found." skip
             "The controls cannot be loaded."
             VIEW-AS ALERT-BOX TITLE "Controls Not Loaded".
&ENDIF

END PROCEDURE.
```

NOTES: You can view but not change this AppBuilder-generated code from the Section Editor.

You might wonder why you cannot load the control with a chained handle reference CtrlFrame:COM-HANDLE:LoadControls(...). The reason is that even though CtrlFrame:COM-HANDLE returns a component handle, it does so with reference to a widget attribute (COM-HANDLE), not a COM object property or method. You cannot make a component handle expression by chaining widget handle references (that return component handles) with component handle references.

9.7.3 Initializing the Control

The programmer creates the initialize-controls procedure if they want to modify control properties before the control becomes visible. In addition to initializing the component handle to the CSSpin control, the example procedure gets the initial setting of the CSSpin Value property. This allows the correct value to display when enable_UI visualizes the Customer Information dialog box for the first time:

```
/* **********************  Internal Procedures  ********************** */
            .
            .
            .
PROCEDURE initialize-controls :
/*------------------------------------------------------------------------
  Purpose:
  Parameters:  <none>
  Notes:
------------------------------------------------------------------------*/

    chCSSpin = chcustSpin:CSSpin.

    /* Must initialize Record_Count from the initial */
    /* spin control value after control is loaded.   */

    Record_Count = chCSSpin:Value.

END PROCEDURE.
```

NOTE: The initialize-controls procedure is the earliest place provided by the AppBuilder Section Editor to dynamically modify properties for UI-enabled ActiveX controls.

After an ActiveX control instance is loaded, you can interact with it in several ways. You have just seen how you might retrieve a property value to display during user interface initialization. The most common interactions with an ActiveX control occur in the Progress event procedures that you write to handle OCX events.

9.7.4 Using Event Procedures

The AppBuilder provides definition templates for event procedures through the Triggers section of the Section Editor. (For more information, see the "Handling Events" section.) In this application, the CSSpin control's SpinUp and SpinDown events are caught to advance or backup in the list of Customer records. These events also increment (SpinUp) or decrement (SpinDown) the spin button control Value property, which the application displays in the Record_Count field.

This is the event procedure for the SpinUp event. As you hold down the right-arrow button of the control, the control generates continuous SpinUp events. For each such event, this procedure reads the next query record and displays it. However, if the procedure tries to read beyond the last record in the query, it resets max-records to 1 less than the current Value property setting (the last record count):

```
/* ************************  Control Triggers  ********************** */
            .
            .
            .
PROCEDURE custSpin.CSSpin.SPINUP .
/*-----------------------------------------------------------------------
  Purpose:
  Parameters Required for this OCX:
   None
  Notes:
-----------------------------------------------------------------------*/

    GET NEXT Dialog-Frame.
    IF NOT AVAILABLE Customer THEN DO:
        /* Control is incremented to last Customer + 1 */
        max-records = chCSSpin:Value - 1.
        chCSSpin:Value = 1.
        GET FIRST Dialog-Frame.
    END.
    RUN displayCustomer.

END PROCEDURE.
```

It also resets the Value property to 1, wrapping around and restarting the upward query scan at the first record. Thus, the user can scan upward continuously through the query without reversing direction.

NOTE: The reason the application calculates a new value for max-records when it passes the last record is to get an updated result if records are deleted after NUM-RESULTS from the query is initially returned. However, note that the value might be inaccurate if records are deleted after the user encounters them. So, this is not a perfect solution.

This is the event procedure for the SpinDown event. As you hold down the left-arrow button of the control, the control generates continuous SpinDown events. For each such event, this procedure reads the previous query record and displays it. However, if the procedure tries to read before the first record in the query, it resets the current Value property setting to the known count of the last record in the query and restarts the downward query scan at the last record:

```
/* ************************  Control Triggers  ********************** */
            .
            .
            .
PROCEDURE custSpin.CSSpin.SPINDOWN .
/*-----------------------------------------------------------------------
  Purpose:
  Parameters Required for this OCX:
   None
  Notes:
-----------------------------------------------------------------------*/

    GET PREV Dialog-Frame.
    IF NOT AVAILABLE Customer THEN DO:
        chCSSpin:Value = max-records.
        GET LAST Dialog-Frame.
    END.
    RUN displayCustomer.

END PROCEDURE.
```

Thus, the user can scan downward continuously through the query without reversing direction.

NOTE: The reason the application needs to set max-records from the number of query entries at the very start of execution (in the Main Block) might already be apparent. If the user begins by scanning downward from the first record, the application now knows the correct value to reset the record counter (Value property) to start the downward query scan at the last record.

9.7.5 Interacting Outside Of Event Procedures

Finally, this is a simple example of where you might reference an ActiveX control from a procedure other than an event procedure. The displayCustomer procedure retrieves a changed setting of the CSSpin Value property and displays it as the current record position for any event that changes that position:

```
/* **********************  Internal Procedures  ********************** */
            .
            .
            .
PROCEDURE displayCustomer :
/*------------------------------------------------------------------------
  Purpose:
  Parameters:  <none>
  Notes:
------------------------------------------------------------------------*/

    Record_Count = chCSSpin:Value.
    DISPLAY sports.Customer.name sports.Customer.Address
        sports.Customer.Address2 sports.Customer.Balance
        sports.Customer.City sports.Customer.Comments
        sports.Customer.Contact sports.Customer.Country
        sports.Customer.Credit-Limit sports.Customer.Cust-Num
        sports.Customer.Discount sports.Customer.Postal-Code
        sports.Customer.Sales-Rep sports.Customer.Terms
        sports.Customer.state sports.Customer.phone
         Record_Count
    WITH FRAME Dialog-Frame.

END PROCEDURE.
```

Other interactions are possible within triggers for control-frame events and by passing the control handle to external and persistent procedures.

9.8 ActiveX Controls and Examples Installed With Progress

Progress comes installed with three ActiveX controls, all of which are licensed for design mode. The following sections provide a brief overview of what each control does. For complete information on each control and its supported properties, methods, and events, see the Progress On-line Reference that comes with your installation or access on-line help from any tool of the Progress Application Development Environment (ADE).

9.8.1 Combo Box Control (CSComboBox)

CSComboBox, a combo box control available through Progress Software Corporation, offers several search modes to locate data in the list box portion of the control. It also provides several ways to populate the list box, both by typing in the data at design time and by programmatically setting the data at runtime.

9.8.2 Spin Button Control (CSSpin)

CSSpin, a spin button control available through Progress Software Corporation, allows the user to enter or edit a numeric value, similar to the 4GL slider widget. Where a slider changes value continually as you move it, a spin button changes value each time your press it or continually as you press and hold it. There are actually two buttons in the control for increase and decrease of value. The Value property of the control can return values from -32768 to 32767, depending on the settings of the Min and Max properties.

You can also use the value-changing events of the CSSpin control (SpinDown and SpinUp) to move back and forth through a database query with or without reference to the Value property. If necessary, you can programmatically reset the Value property to spin through more query rows than the integer range of the Value property might allow. For an example of query navigation provided by the CSSpin control, see the `e-ocx1.w` example procedure described in this chapter.

9.8.3 Timer Control (PSTimer)

PSTimer, a control available through Progress Software Corporation, allows you to program tasks that execute at regular time intervals. You execute these tasks in an event procedure for the PSTimer Tick event. The Tick event fires at a programmable time interval that you can set using the Interval property.

9.8.4 Example Applications Using ActiveX Controls

In addition to the on-line example, `e-ocx1.w`, Progress comes installed with a number of sample applications using ActiveX controls that you can use to test and borrow code for your own application development. These applications reside in separate subdirectories under `%DLC%\src\samples\ActiveX`. Each subdirectory contains a set of files for one application. These files include a `readme.txt` file that describes the requirements for running the application and the capabilities that it demonstrates.

10

Sockets

The 4GL provides direct access to TCP/IP sockets. Sockets provide a means to implement interprocess communications with both local and remote processes. Using 4GL sockets, you can communicate with non-4GL processes, as well as other 4GL processes. Thus, you can implement socket-based applications completely in the 4GL that otherwise require the use of C modules accessible only through the HLC or shared library interfaces.

This chapter describes how you can access TCP/IP sockets directly from the 4GL. It assumes that you are familiar with basic TCP/IP socket programming.

This chapter contains the following sections:

10.1 4GL For Programming Sockets

Table 10–1 lists the 4GL elements that are either valid only for working with sockets or have special application in socket programming. The remaining sections in this chapter explain how to use these elements.

Table 10–1: 4GL For Programming Sockets *(1 of 7)*

4GL Element	Description
BYTES-READ	An INTEGER attribute on the socket object handle that returns the number of bytes read during the last invocation of the socket READ( ) method. If the last READ( ) method call on the socket failed, this attribute returns 0.
BYTES-WRITTEN	An INTEGER attribute on the socket object handle that returns the number of bytes written during the last invocation of the socket WRITE( ) method. If the last WRITE( ) method call on the socket failed, this attribute returns 0.
CONNECT	An event received on a server socket object handle that indicates that a socket client is trying to connect. This event, if handled by an I/O-blocking or PROCESS EVENTS statement, executes any CONNECT event procedure defined for the server socket object. You can use this event procedure to obtain the socket object with which the client is communicating.
CONNECT (*connection-parameters*)	A method on the socket object handle that connects a socket handle to a specified TCP/IP port on a specified host.
CONNECTED( )	A method on the socket object handle that indicates if a socket handle is currently connected to a port.
CREATE SERVER-SOCKET *server-socket-handle*	A statement that creates a server socket object with all attributes set to their default values, and stores its handle in a HANDLE variable.
CREATE SOCKET *socket-handle*	A statement that creates a socket object with all attributes set to their default values and stores its handle in a HANDLE variable.

Table 10–1: 4GL For Programming Sockets *(2 of 7)*

4GL Element	Description
DEFINE INPUT PARAMETER *socket-handle*	A statement that defines the single INPUT parameter to the CONNECT event procedure (specified using the SET-CONNECT-PROCEDURE( ) method). This parameter returns the handle to the socket object created when a socket server receives a CONNECT event, and which the socket server uses to communicate with the corresponding socket client.
DELETE OBJECT *handle*	A statement that you can use to delete a handle, including socket and server socket object handles. To delete a connected socket object, you must first disconnect it using the DISCONNECT( ) method. To delete a server socket object enabled to listen for connections, you must first disable it using the DISABLE-CONNECTIONS( ) method.
DISABLE-CONNECTIONS( )	A method on the server socket object handle that indicates that new connections are no longer accepted on the server socket.
DISCONNECT( )	A method on the socket object handle that terminates the connection between the socket object and the port to which it is connected.
ENABLE-CONNECTIONS (*connection-parameters*)	A method on the server socket object handle that specifies the TCP/IP port that Progress uses to listen for new connections. Once called, Progress automatically listens for and accepts new connections on the specified port. ENABLE-CONNECTIONS() also lets you specify the length of the pending-connection queue.
FIRST-SERVER-SOCKET	A HANDLE attribute on the SESSION object handle that returns the handle to the first entry in the chain of server socket handles for the session. Note that you can have only one server socket object in the list enabled to listen for events at one time.

Table 10–1: 4GL For Programming Sockets *(3 of 7)*

4GL Element	Description
FIRST-SOCKET	A HANDLE attribute on the SESSION object handle that returns the handle to the first entry in the chain of socket handles for the session.
GET-BYTES-AVAILABLE( )	A method on the socket object handle that indicates the number of bytes available for reading from the socket.
GET-SOCKET-OPTION (*option-name*)	A method on the socket object handle that returns the specified TCP socket option. Progress supports the following options: • TCP-NODELAY • SO-LINGER • SO-KEEPALIVE • SO-REUSEADDR • SO-SNDBUF • SO-RCVBUF • SO-RCVTIMEO For more information on these options, see the *Progress Language Reference* and your TCP documentation.
LAST-SERVER-SOCKET	A HANDLE attribute on the SESSION object handle that returns the handle to the last entry in the chain of server socket handles for the session. Note that you can have only one server socket object in the list enabled to listen for events at one time.
LAST-SOCKET	A HANDLE attribute on the SESSION object handle that returns the handle to the last entry in the chain of socket handles for the session.
LOCAL-HOST	A CHARACTER attribute on the socket object handle that returns the IP address on the local machine where the socket object is connected.

Table 10–1: 4GL For Programming Sockets *(4 of 7)*

4GL Element	Description
LOCAL-PORT	An INTEGER attribute on the socket object handle that returns the local port number of the socket object.
MEMPTR	The Progress data type used by the socket handle READ( ) and WRITE( ) methods to read and write data on a socket. A MEMPTR expression defines a memory region whose size you must allocate in bytes. MEMPTR functions and statements allow you to read and write data between most other Progress data types and the specified memory region. For more information the MEMPTR data type, see Chapter 1, "Introduction."
NEXT-SIBLING	A HANDLE attribute on the socket and server socket object handle that returns the next entry in the list of socket or server socket handles created for the current Progress session.
PREV-SIBLING	A HANDLE attribute on the socket and server socket object handle that returns the previous entry in the list of socket or server socket handles created for the current Progress session.
PROCESS EVENTS	A statement that you can use to handle any pending CONNECT or READ-RESPONSE events. You can also use any I/O-blocking statement, such as the WAIT-FOR statement.
READ(*memptr-expression* , *position* , *bytes-to-read* [, *mode*])	A method on the socket object handle that reads data from the specified socket. The method specifies the MEMPTR memory region and byte position within the region to store the data, a number of bytes to read (and store) from the socket, and a read mode. The read mode indicates if the exact specified number of bytes must be read or up to the specified number of bytes can be read.

Table 10–1: 4GL For Programming Sockets *(5 of 7)*

4GL Element	Description
READ-RESPONSE	An event received on a socket object handle indicating that data is waiting on the socket to be read. This event, if handled by an I/O-blocking or PROCESS EVENTS statement, executes any READ-RESPONSE event procedure defined for the socket object. You can use this event procedure if you want to read data from the socket in an event-driven manner.[1]
REMOTE-HOST	A CHARACTER attribute on the socket object handle that returns the IP address of the remote machine with which a connected socket object is communicating.
REMOTE-PORT	An INTEGER attribute on the socket object handle that returns the number of the port on the remote machine with which a connected socket object is communicating.
SELF	A system handle that returns the handle of the object on which an event is handled in the context of an event procedure. In a CONNECT event procedure, this is the handle to the server socket that is responding to a CONNECT event. In a READ-RESPONSE event procedure, this is the handle to the socket that is responding to a READ-RESPONSE event.
SENSITIVE	A LOGICAL attribute on the socket and server socket object handle that indicates whether the object can receive events. Set to TRUE (receive events) by default.
Server Socket Object Handle	A handle to a server socket object. This object allows you to listen for and accept TCP/IP connections on a given port.
SET-CONNECT-PROCEDURE(*event-internal-procedure* [, *procedure-context*])	A method on the server socket object handle that specifies the name of an internal procedure (*CONNECT event procedure*) to invoke when a CONNECT event occurs.

Table 10–1: 4GL For Programming Sockets *(6 of 7)*

4GL Element	Description
SET-READ-RESPONSE -PROCEDURE(*event-internal-procedure* [, *procedure-context*])	A method on the socket object handle that specifies the name of an internal procedure (*READ-RESPONSE event procedure*) to invoke when a READ-RESPONSE event occurs.
SET-SOCKET-OPTION (*name*, *arguments*)	A method on the socket object handle that sets the specified TCP socket option. Progress supports the following options: • TCP-NODELAY • SO-LINGER • SO-KEEPALIVE • SO-REUSEADDR • SO-SNDBUF • SO-RCVBUF • SO-RCVTIMEO For more information on these options, see the *Progress Language Reference* and your TCP documentation.
Socket Object Handle	A handle to a socket object. This object allows you to read or write data on a TCP/IP socket and to perform other TCP/IP socket actions.
TYPE	A CHARACTER attribute on the socket and server socket object handle that returns the handle type, which is “SERVER-SOCKET” for a server socket handle and “SOCKET” for a socket handle.
WAIT-FOR ...	A statement that you can use to handle any pending CONNECT or READ-RESPONSE events. You can also use PROCESS EVENTS or any other I/O-blocking statement, such as the PROMPT-FOR statement, to handle the events. When a CONNECT or READ-RESPONSE event occurs in the context of these statements, any CONNECT or READ-RESPONSE event procedure specified for the corresponding handle is executed.

Table 10–1: 4GL For Programming Sockets *(7 of 7)*

4GL Element	Description
WRITE(*memptr-expression* , *position* , *bytes-to-write*)	A method on the socket object handle that writes data to the specified socket. The method specifies the MEMPTR memory region and byte position within the region from which to write the data, as well as the number of bytes to write from the region.

[1] A socket handle receives a READ-RESPONSE event under the same conditions that cause the TCP/IP select function to indicate that a socket is ready to receive results. However, in the 4GL, you must read data on the socket to continue to receive the event. For more information, see the "Data Detection Using the Event-driven Model" section.

10.2 Overview Of Tasks To Implement 4GL Sockets

Socket programming can be typically divided into three parts:

- Server programming tasks
- Client programming tasks
- Tasks common to both clients and servers

This section summarizes the tasks for programming sockets in the 4GL using the same framework. The sections that follow describe each of these tasks in more detail.

Table 10–2 lists socket programming tasks by application function, indicates whether each task is required for implementing socket communications, and provides a reference to a following section for more information on the task.

Table 10–2: Summary Of Socket Programming Tasks *(1 of 2)*

Application Function	Tasks
Socket Server	• Create a server socket object and enable it to listen for connections (CONNECT events) on a specified port (required). See the "Implementing a 4GL Socket Server" section. • Define a CONNECT event procedure to respond to CONNECT events and return the socket object handle created for each connection (required). See the "Listening and Responding To Connection Requests" section. • Make the server socket temporarily insensitive to CONNECT events. See the "Managing the Server Socket" section. • Disable the server socket from listening for new socket connections. See the "Managing the Server Socket" section. • Delete the server socket object. See the "Managing the Server Socket" section.

Table 10–2: Summary Of Socket Programming Tasks *(2 of 2)*

Application Function	Tasks
Socket Client	• Create a socket object to read and write data, and connect it to a port on a socket server (required). See the "Implementing a 4GL Socket Client" section.
Client or Server	• Define and initialize a MEMPTR variable (required to read or write data on a socket). See the "Defining and Initializing MEMPTR Variables" section. • Test whether data is available to read on a socket. See the "Data Detection Using the Procedural Model" section. • Define a READ-RESPONSE event procedure to respond to data coming in on a socket connection. See the "Data Detection Using the Event-driven Model" section. • Read data on a socket and test for the number of bytes read for each read operation. See the "Reading Data On a Socket" section. • Write data on a socket and test for the number of bytes written for each write operation. See the "Writing Data On a Socket" section. • Marshall data into a MEMPTR buffer for writing on a socket and unmarshall data from a MEMPTR buffer after reading on a socket. See the "Marshalling and Unmarshalling Data For Socket I/O" section. • Test whether a socket object is connected. See the "Managing Sockets and Their Connections" section. • Make a socket temporarily insensitive to READ-RESPONSE events. See the "Managing Sockets and Their Connections" section. • Set and get socket options. See the "Managing Sockets and Their Connections" section. • Obtain the local and remote host IP addresses and port numbers involved in a socket connection. See the "Managing Sockets and Their Connections" section. • Disconnect a socket object from its remote host and port. See the "Managing Sockets and Their Connections" section. • Delete socket objects. See the "Managing Sockets and Their Connections" section.

10.3 Implementing a 4GL Socket Server

To implement a socket server, follow these steps:

1 ♦ Create a server socket object using the CREATE SERVER-SOCKET statement.

2 ♦ Enable the server socket to listen for connections using the ENABLE-CONNECTIONS( ) method. This assigns the socket to a specified TCP/IP port on which Progress listens and accepts connection requests.

NOTE: ENABLE-CONNECTIONS() also lets you set the length of the pending-connection queue. For more information, see the reference entry for the ENABLE-CONNECTION() method in the *Progress Language Reference*.

3 ♦ Specify a CONNECT event procedure using the SET-CONNECT-PROCEDURE( ) method.

4 ♦ Wait for CONNECT events using a blocking I/O statement or poll for CONNECT events using the PROCESS EVENTS statement.

Once you have a connection established from a client, you can read data from and write data to the client on the connection. You can also manage the server socket that you have enabled for listening.

The following sections describe these steps in more detail.

10.3.1 Listening and Responding To Connection Requests

After you enable a server socket for listening using the ENABLE-CONNECTIONS( ) method, Progress listens on the specified port for client connections. When a socket client sends a connection request to this port, Progress automatically accepts the connection.

After accepting the connection, Progress posts a CONNECT event on the server socket and calls the CONNECT event procedure that you have specified using the SET-CONNECT-PROCEDURE( ) method. The CONNECT event procedure executes, returning a handle to a socket object created by Progress and passed as a parameter to the event procedure context. This socket object, then, is the communications endpoint for the connection on the server.

Thus, to listen for connections on a server socket using the CONNECT event procedure:

1 ♦ Define an internal procedure that takes one INPUT parameter of type HANDLE to serve as an event procedure. You can define this procedure in any procedure context that is active while listening for connections.

2 ♦ Specify the procedure you defined in Step 1 as a CONNECT event procedure by invoking the SET-CONNECT-PROCEDURE( ) method on the server socket handle that you have enabled for listening.

3 ♦ Include blocking I/O statements (such as WAIT-FOR) or PROCESS EVENTS statements in your code to handle events. When any CONNECT event is received in the context of one of these statements, the event procedure specified in Step 2 executes.

Once you have the handle to a connected socket object, you can read and write data on the socket, and otherwise manage the socket for the connection. For more information, see the "Read, Writing, and Managing Sockets On Clients and Servers" section.

10.3.2 Managing the Server Socket

After you create and enable a server socket to listen for and accept connections, you can perform the following management functions on the server socket:

- **Control event sensitivity** — At any time, you can make the server socket stop receiving CONNECT events by setting its SENSITIVE attribute to FALSE. You can, at any time, return it to listen for CONNECT events by setting SENSITIVE to TRUE. Thus, when SENSITIVE is FALSE, Progress still listens on the specified port, but temporarily stops accepting connections and posting CONNECT events.
- **Disable listening for connections** — You can permanently stop the server socket from accepting connections by invoking its DISABLE-CONNECTIONS( ) method. This stops the server socket from listening on the current port for new connections. However, all currently-connected sockets remain connected. If a client attempts to connect on the port, it receives an error.
- **Delete the server socket object** — You can delete a server socket object using the DELETE OBJECT statement. However, you must disable the server socket from listening for and accepting connections before you can delete it.

10.4 Implementing a 4GL Socket Client

To implement a socket client, follow these general steps:

1 ♦ Create a socket object using the CREATE SOCKET statement.

2 ♦ Connect the socket object to the host and listening port of a socket server using the CONNECT( ) method on the socket handle.

3 ♦ Read and write data on the socket and otherwise manage the socket.

The tasks required to read and write data on a socket and to manage the socket are identical for both socket clients and socket servers. For more information, see the "Read, Writing, and Managing Sockets On Clients and Servers" section.

10.5 Read, Writing, and Managing Sockets On Clients and Servers

To read and write data on a socket object, you can use one of the following models, or a combination of both:

- **Procedural model** — Where you poll for and read data wherever necessary using the READ( ) method on the socket handle and write data wherever necessary using the WRITE( ) method on the socket handle.
- **Event-driven model** — Where you wait for a READ-RESPONSE event on the socket handle. You then read and write whatever data you need on the socket handle within an event procedure that executes in response to this event.

Regardless of the model you use, you must first define and initialize a MEMPTR variable to hold the data you want to read or write.

10.5.1 Defining and Initializing MEMPTR Variables

As with any 4GL variable, you define a MEMPTR variable using the DEFINE VARIABLE statement. Unlike most other data types, MEMPTR variables also have two other features that you must initialize to use them for socket communications:

- The size of the memory region reserved for the variable
- The byte order that Progress is to use to interpret certain data types that you can store in the variable

Before you can read or write to a MEMPTR variable, you must set the size of memory reserved for it, in bytes, using the 4GL SET-SIZE statement. Depending on your data and application, you might also have to specify the byte order for reading and writing.

Generally, when writing and reading data on a socket, the client and server must agree on the byte order so Progress can consistently interpret, on both ends, the order of bytes that comprise each data type stored in the MEMPTR region. Thus, the MEMPTR data type supports the SET-BYTE-ORDER statement and the GET-BYTE-ORDER function to set and get the byte order for a MEMPTR variable. For more information, see Chapter 1, “Introduction.”

10.5.2 Detecting Data On a Socket

You can detect data on a socket using either the procedural or event-driven models.

Data Detection Using the Procedural Model

To detect data procedurally, you can simply read whatever data is available on the socket object using the READ( ) method. Using this model, you do not use the READ-RESPONSE event, and Progress does not automatically notify your application when data is available to read on the socket. You can also:

- Check whether the socket is connected using the CONNECTED( ) method.
- Check for how many bytes are available to read, using the GET-BYTES-AVAILABLE( ) method.

How you use these methods depends on your application requirements and the options that you choose on the READ( ) method. For more information on READ( ) method options, see the “Reading Data On a Socket” section.

Data Detection Using the Event-driven Model

To detect data using events, you set up an event handler for READ-RESPONSE events posted on a connected socket object. You can set up this event handler as follows:

1 ♦ Define an internal procedure that takes no arguments to serve as an event procedure. You can define this procedure in any procedure context that is active during the connection. When this procedure executes in response to a READ-RESPONSE event, the SELF system handle returns the handle of the connected socket.

2 ♦ Specify the procedure you defined in Step 1 as a READ-RESPONSE event procedure by invoking the SET-CONNECT-PROCEDURE( ) method on the socket.

3 ♦ Include blocking I/O statements (such as WAIT-FOR) or PROCESS EVENTS statements in your code to initiate the handling of events. When any READ-RESPONSE event is received in the context of one of these statements, the event procedure specified in Step 2 executes.

In the event-driven model, Progress can post a READ-RESPONSE event on a connected socket object for two reasons:

- **Data has arrived on the socket** — You can then read this data during execution of the event procedure (or any time while the socket remains connected) using the READ( ) method. You do not have to read all the bytes available on the socket. If any data remains after reading on the socket, Progress immediately posts another READ-RESPONSE event on the socket object for the available data.

  NOTE: After Progress posts the first READ-RESPONSE event for a new socket, Progress does not post another READ-RESPONSE event for the socket until you call the READ( ) method on the socket object. Thus, if you do not read data on the socket in the event procedure, you must make sure to do so elsewhere in your application if you want the application to respond to any additional events for the socket.

- **The socket is disconnected** — During execution of an event procedure called for a socket disconnection:

  – Calling the READ( ) method on the socket object returns an error.

  – The value returned by a GET-BYTES-AVAILABLE( ) method invoked on the socket object is zero (0).

  – The value returned by a CONNECTED( ) method invoked on the socket object is FALSE.

10.5.3 Reading Data On a Socket

To read data from a connected TCP/IP socket, you invoke the READ( ) method on the corresponding socket object (SELF:READ( ) within the corresponding READ-RESPONSE event procedure). You can invoke this method on a connected socket at any time to read data. However, the method blocks depending on the amount of data available on the socket, the reading mode that you use, and the timeout value (set by the SO-RCVTIMEO option of the SET-SOCKET-OPTION() method).

The syntax of the READ( ) method follows:

SYNTAX

```
READ(
      memptr-expression
    , position
    , bytes-to-read
      [ , mode ]
    )
```

The READ( ) method transfers data from the socket to the specified MEMPTR variable, *memptr-expression*, at the byte position in the MEMPTR region specified by the INTEGER expression, *position*. Exactly how much data the READ( ) method blocks to read depends on:

- The number of bytes specified by the INTEGER *bytes-to-read*
- The reading mode specified by the INTEGER *mode*
- The timeout value, set by the SO-RCVTIMEO option of the SET-SOCKET-OPTION() method

Specifying the Read Mode

You can specify *mode* using a compiler constant as shown in Table 10–3.

Table 10–3: Socket Reading Modes

Compiler Constant	Value	Description
READ-AVAILABLE	1	The READ( ) method blocks until at least one (1) byte is available on the socket and reads the number of bytes that are available up to a maximum of *bytes-to-read* bytes.
READ-EXACT-NUM	2	(Default) The READ( ) method blocks until *bytes-to-read* bytes are available to read from the socket.

Thus, you can have the READ( ) method block until exactly the specified number of bytes are read (the default), or until all available bytes are read up to a maximum number allowed.

The appropriate reading mode to use depends on your application requirements. Note, however, that if you specify READ-EXACT-NUM, the READ( ) method blocks until it reads the specified number of bytes (no matter how long it takes) or until the socket is disconnected (whatever happens first).

Specifying the Timeout Length

Besides setting the read mode, you can also set the amount of time READ() waits before timing out. To do so, use the SO-RCVTIMEO option of the SET-SOCKET-OPTION() method. If you do not set a timeout value, the default is for READ() to wait forever.

READ()'s timeout behavior is affected by the interaction of the read mode and the timeout value, as Table 10–4 illustrates.

Table 10–4: Effect Of Read Mode and Timeout Value On READ()'s Timeout Behavior

Read Mode	With a Timeout Value	Without a Timeout Value
READ-AVAILABLE	Assuming there are no connection failures, READ() blocks until one for the following occurs: • The timeout expires. • There is at least one byte available to read on the socket.	Assuming there are no connection failures, READ() blocks until there is at least one byte available to read on the socket.
READ-EXACT-NUM	Assuming there are no connection failures: If there is any data on the socket, READ() blocks until there are *bytes-to-read* bytes available to read on the socket. Else, READ() blocks until the timeout expires.	Assuming there are no connection failures, READ() blocks until there are *bytes-to-read* bytes available to read on the socket.

Verifying the Number Of Bytes Actually Read

You can verify the number of bytes actually read by the READ( ) method. The number of bytes read by the last READ( ) method invoked on a socket object is equal to the value of the BYTES-READ attribute invoked on the same socket object.

10.5.4 Writing Data On a Socket

You can write data on a connected TCP/IP socket at any time using the WRITE( ) method on the corresponding socket object (SELF:WRITE( ) within the corresponding READ-RESPONSE event procedure).

The syntax of the WRITE( ) method follows:

SYNTAX

```
WRITE(
      memptr-expression
    , position
    , bytes-to-write
    )
```

The WRITE( ) method transfers the number of bytes specified by the INTEGER expression, *bytes-to-write*, from the specified MEMPTR variable, *memptr-expression*, to the socket starting from the byte position within the MEMPTR region specified by the INTEGER expression, *position*.

You can verify the number of bytes actually written by the WRITE( ) method. The number of bytes written by the last WRITE( ) method invoked on a socket object is equal to the value of the BYTES-WRITTEN attribute invoked on the same socket object.

10.5.5 Marshalling and Unmarshalling Data For Socket I/O

After reading data from a socket into a MEMPTR variable, any 4GL procedure that needs to interpret the data must unmarshall the data from the MEMPTR memory region. Unmarshalling the data converts it from bytes in memory to the 4GL data types that the 4GL procedure can use, as determined by your application and the byte order of the MEMPTR data.

Similarly, before writing data to a socket from a MEMPTR variable, a 4GL procedure must marshall data into the MEMPTR memory region. Marshalling the data converts it from the 4GL data types the procedure understands to bytes in memory that can be written to the socket. Again, how you organize the bytes in the MEMPTR memory region depends on your application and the application you are communicating with.

The 4GL supports several statements and functions for marshalling and unmarshalling data in different forms. All such statements and functions that interact directly with MEMPTR variables convert between bytes and 4GL data types according to the MEMPTR byte order that you specify. Before using a MEMPTR variable to read or write socket data, both the socket client and socket server must set their respective MEMPTR variables to an identical byte order. When marshalling and unmarshalling the data, you must make certain to access MEMPTR data in conformance with the agreed and specified MEMPTR byte order.

For more information on byte order and the statements and functions for marshalling and unmarshalling MEMPTR data, see Chapter 1, “Introduction.”

10.5.6 Managing Sockets and Their Connections

To manage a socket and its connection, the 4GL allows you to:

- **Test if the socket is connected** — If the CONNECTED( ) socket method is TRUE, the socket is connected.
- **Control event sensitivity** — At any time, you can make the socket stop receiving READ-RESPONSE events by setting its SENSITIVE attribute to FALSE. You can, at any time, return it to receiving READ-RESPONSE events by setting SENSITIVE to TRUE.
- **Set socket options** — Progress supports the following options:
  - TCP-NODELAY
  - SO-LINGER
  - SO-KEEPALIVE
  - SO-REUSEADDR
  - SO-SNDBUF
  - SO-RCVBUF
  - SO-RCVTIMEO

  To set and get these options, use the SET-SOCKET-OPTION( ) and GET-SOCKET-OPTION( ) methods.

- **Obtain host and port values** — You can obtain the remote host IP address and port number involved in the connection from the values of the REMOTE-HOST and REMOTE-PORT socket attributes. Similarly, you can obtain the local host and port from the LOCAL-HOST and LOCAL-PORT socket attributes.

- **Disconnect a socket** — You can close the socket and remove the association between the socket object and its remote port by invoking the DISCONNECT( ) socket method. Progress automatically closes the local socket when Progress detects that the corresponding remote socket in a connection is closed.

- **Delete the socket object** — You can delete a socket object using the DELETE OBJECT statement. However, you must disconnect the socket before you can delete it.

10.6 Examples Using 4GL Sockets

The following sample procedures show a simple interaction between a socket server (`e-sktsv1.p`) and a socket client (`e-sktcl1.p`). In this case, the socket server exits after handling a single connection and the socket client exits after receiving one data transmission from the server. These are event-driven examples, where the socket server only writes and the socket client only reads on their respective sockets:

e-sktsv1.p

```
DEFINE VARIABLE hServerSocket AS HANDLE.
DEFINE VARIABLE aOk AS LOGICAL.
DEFINE VARIABLE mBuffer AS MEMPTR.
DEFINE VARIABLE greeting AS CHAR.

MESSAGE "We are in the socket server".
CREATE SERVER-SOCKET hServerSocket.

/* Marshal data to write */
SET-SIZE(mBuffer) = 64.
greeting = "SERVER - hello".
PUT-STRING(mBuffer,1) = greeting.

MESSAGE "Start event handling".
hServerSocket:SET-CONNECT-PROCEDURE( "connProc").
aOk = hServerSocket:ENABLE-CONNECTIONS( "-S 3333").
MESSAGE "Enabled connections:" aOk.

IF NOT aOk THEN
   RETURN.

/* This WAIT-FOR completes after the first connection */
WAIT-FOR CONNECT OF hServerSocket.

MESSAGE "Freeing up resources".
hServerSocket:DISABLE-CONNECTIONS().
DELETE OBJECT hServerSocket.
MESSAGE "Finished".

PROCEDURE connProc.
  /* Connection procedure for server socket */
  DEFINE INPUT PARAMETER hSocket AS HANDLE.  /*Socket implicitly created*/

  MESSAGE "We are in CONNECT event procedure, connProc".
  hSocket:WRITE (mBuffer,1,LENGTH(greeting)).
  MESSAGE hSocket:BYTES-WRITTEN "bytes written".
END.
```

s-sktcl1.p

```
DEFINE VARIABLE hSocket AS HANDLE.
DEFINE VARIABLE aOk AS LOGICAL.
DEFINE VARIABLE mBuffer AS MEMPTR.
DEFINE VARIABLE cString AS CHARACTER.

MESSAGE "We are in socket client".
CREATE SOCKET hSocket.

hSocket:CONNECT ("-H localhost -S 3333").
IF hSocket:CONNECTED() THEN
  MESSAGE "Connected OK".
ELSE DO:
  MESSAGE "Could not connect".
  RETURN.
END.

/* Do READ-RESPONSE event handling */
MESSAGE "Start event handling".
hSocket:SET-READ-RESPONSE-PROCEDURE( "readProc").

/* This WAIT-FOR completes after the first data reception */
WAIT-FOR READ-RESPONSE OF hSocket.

MESSAGE "Freeing up resources".
hSocket:DISCONNECT().
DELETE OBJECT hSocket.
MESSAGE "Finished".

PROCEDURE readProc.
  /* Read procedure for socket */
  DEFINE VARIABLE mBuffer AS MEMPTR.

  MESSAGE "We are in READ-RESPONSE event procedure, readProc".
  SET-SIZE(mBuffer) = 64.
  SELF:READ (mBuffer,1,SELF:GET-BYTES-AVAILABLE()).
  MESSAGE SELF:BYTES-READ "bytes read".
  cString = GET-STRING(mBuffer,1).  /*Unmarshal data*/
  DISPLAY cString FORMAT "x(64)".
END PROCEDURE.
```

The following sample procedures show how you can transfer a database record between a socket server (e-sktsv2.p) and a socket client (e-sktcl2.p) using a raw transfer. It uses the sports database as the source (server) and a copy of the sports database as the target (client).

This is a simple example in which the server waits for connections indefinitely (until you press the **STOP** key), but always sends the same database record to the client for each CONNECT event. Note how the server uses the RAW transfer to first copy the record from the database, then to specify the size of the MEMPTR memory from which the record is written on the socket. The server stores the size of the record as the first piece of information sent to the client, followed by the record:

e-sktsv2.p *(1 of 2)*

```
/* Sends a new customer record from the sports  database through a socket. */
DEFINE VARIABLE hServerSocket AS HANDLE NO-UNDO.
DEFINE VARIABLE aOk AS LOGICAL NO-UNDO.
DEFINE VARIABLE mBuffer AS MEMPTR NO-UNDO.
DEFINE VARIABLE r1 AS RAW NO-UNDO.
DEFINE VARIABLE len AS INTEGER NO-UNDO.
DEFINE VARIABLE custNum AS INTEGER NO-UNDO.

MESSAGE "We are in the raw transfer socket server".

/* Create a new customer record with a unique Cust-num */
FIND LAST customer.
custNum = customer.Cust-num + 1.
CREATE customer.
customer.Cust-num = custNum.
customer.Name = "Jack Sprat".
customer.Address = "222 Ferdinand St".
customer.City = "Woburn".
customer.State = "MA".
customer.Postal-Code = "01801".

CREATE SERVER-SOCKET hServerSocket.
hServerSocket:SET-CONNECT-PROCEDURE("connProc", THIS-PROCEDURE).

/* Put customer record into a RAW variable and store in local DB. */
RAW-TRANSFER customer TO r1.
RELEASE customer.
```

e-sktsv2.p *(2 of 2)*

```
/* Put the length of the record followed by a copy of the full
   record into a MEMPTR to send it through the socket.
*/
len = LENGTH(r1).
SET-BYTE-ORDER(mBuffer) = LITTLE-ENDIAN.
SET-SIZE(mBuffer) = len + 4.
PUT-LONG(mBuffer, 1) = len.
PUT-BYTES(mBuffer, 5) = r1.

aOk = hServerSocket:ENABLE-CONNECTIONS( "-S 3334").
MESSAGE "Enabled connections:" aOk.

DO ON STOP UNDO, LEAVE:
  WAIT-FOR CLOSE OF THIS-PROCEDURE.
end.

PROCEDURE connProc.
  /* Connection procedure for server socket */
  DEFINE INPUT PARAMETER hSocket AS HANDLE.  /*Socket implicitly created*/
  MESSAGE "We are in the CONNECT event procedure, connProc".

  /* Send the size followed by the record that we put into the MEMPTR */
  hSocket:WRITE (mBuffer,1, GET-SIZE(mBuffer)).
  MESSAGE "Sent" hSocket:BYTES-WRITTEN "bytes containing customer record".
END.
```

In this example, the client (`e-sktcl2.p`) polls its socket procedurally until the data for the record is available to read. In this case, the client first waits for the size information, then waits for that number of bytes of customer data. It also uses this size information to set the size of the MEMPTR region for reading the record off the socket. Finally, note that the client deletes the socket object and frees MEMPTR memory after it disconnects from the server:

e-sktcl2.p

```
/* Receives a new customer record through a socket and puts it into the
   sports DB.
*/
DEFINE VARIABLE hSocket AS HANDLE NO-UNDO.
DEFINE VARIABLE mBuffer AS MEMPTR NO-UNDO.
DEFINE VARIABLE r1 AS RAW NO-UNDO.
DEFINE VARIABLE recLen AS INTEGER NO-UNDO.

MESSAGE "We are in the raw transfer socket client".

CREATE SOCKET hSocket.
hSocket:CONNECT ("-H localhost -S 3334").
IF hSocket:CONNECTED() THEN
  MESSAGE "Connected OK".
ELSE DO:
  MESSAGE "Could not connect".
  RETURN.
END.

/* Do a blocking READ until we get the count of how long the
   record is to read.
*/
SET-SIZE(mBuffer) = 4.
SET-BYTE-ORDER(mBuffer) = LITTLE-ENDIAN.
hSocket:READ (mBuffer, 1, 4).
recLen = GET-LONG(mBuffer, 1).

/* READ again to get the full record. */
SET-SIZE(mBuffer) = 0.
SET-SIZE(mBuffer) = recLen.
hSocket:READ (mBuffer, 1, recLen).
MESSAGE hSocket:BYTES-READ "bytes read for customer record".

/* Copy the record to a RAW variable so we can put it into the DB */
r1 = mBuffer.
RAW-TRANSFER r1 TO customer.

MESSAGE "Customer is:" Cust-num Name Address City State Postal-Code.
RELEASE customer.

hSocket:DISCONNECT().
DELETE OBJECT hSocket.
SET-SIZE(mBuffer) = 0.
```

The following example also involves a client and server.

The server, e-sktsv3.p, demonstrates how to set qsize, the length of the pending-connection queue, while enabling the server-socket for connections.

e-sktsv3.p

```
DEFINE VARIABLE sserver AS HANDLE.
DEFINE VARIABLE ok AS LOGICAL.
DEFINE VARIABLE m-string AS MEMPTR.

CREATE SERVER-SOCKET sserver.
ok = sserver:SET-CONNECT-PROCEDURE( "connProc").
MESSAGE "set connection procedure" ok.
```

```
/* Enable for connections and set the qsize */
ok = sserver:ENABLE-CONNECTIONS( "-S 3000 -qsize 5").
MESSAGE "enable connections" OK.

WAIT-FOR CONNECT OF sserver.

PROCEDURE connProc.
/* connection procedure for server socket */
DEFINE INPUT PARAMETER hsocket AS HANDLE.  /*implicitly created*/
```

```
MESSAGE "Inside connProc".
SET-SIZE(m-string) = 64.
PUT-STRING(m-string,1) = "SERVER - hello".
ok = hsocket:WRITE (m-string,1,26).
END.
```

The client, e-sktcl3.p, shows how to set and retrieve socket options.

e-sktcl3.p *(1 of 2)*

```
DEFINE VARIABLE hsocket AS HANDLE.
DEFINE VARIABLE ok AS LOGICAL.
DEFINE VARIABLE ret AS CHARACTER.
DEFINE VARIABLE m-string AS MEMPTR.
DEFINE VARIABLE c AS CHARACTER.
```

```
CREATE SOCKET hsocket.

hsocket:CONNECT ("-H localhost -S 3000 ").
IF hsocket:CONNECTED() THEN
    MESSAGE "Connected OK".
ELSE DO:
    MESSAGE "Could not connect".
    RETURN.
END.
```

```
/* connect succeeded */

OK = hsocket:SET-SOCKET-OPTION("TCP-NODELAY", "TRUE").
ret = hsocket:GET-SOCKET-OPTION("TCP-NODELAY").
MESSAGE "TCP-NODELAY = " ret.
```

```
OK = hsocket:SET-SOCKET-OPTION("SO-LINGER", "TRUE, 55").
ret = hsocket:GET-SOCKET-OPTION("SO-LINGER").
MESSAGE "SO-LINGER = " ret.
```

```
OK = hsocket:SET-SOCKET-OPTION("SO-KEEPALIVE", "TRUE").
ret = hsocket:GET-SOCKET-OPTION("SO-KEEPALIVE").
MESSAGE "SO-KEEPALIVE = " ret.
```

```
OK = hsocket:SET-SOCKET-OPTION("SO-REUSEADDR", "TRUE").
ret = hsocket:GET-SOCKET-OPTION("SO-REUSEADDR").
MESSAGE "SO-REUSEADDR = " ret.
```

```
OK = hsocket:SET-SOCKET-OPTION("SO-SNDBUF", "40").
ret = hsocket:GET-SOCKET-OPTION("SO-SNDBUF").
MESSAGE "SO-SNDBUF = " ret.
```

```
OK = hsocket:SET-SOCKET-OPTION("SO-RCVBUF", "5000").
ret = hsocket:GET-SOCKET-OPTION("SO-RCVBUF").
MESSAGE "SO-RCVBUF = " ret.
```

```
OK = hsocket:SET-SOCKET-OPTION("SO-RCVTIMEO", "60").
ret = hsocket:GET-SOCKET-OPTION("SO-RCVTIMEO").
MESSAGE "SO-RCVTIMEO = " ret.
```

e-sktcl3.p *(2 of 2)*

```
WAIT-FOR READ-RESPONSE OF hsocket.
```

```
SET-SIZE(m-string) = 64.
ok = hsocket:READ (m-string,1,26).
c = GET-STRING(m-string,1).
MESSAGE "the string = " c.
```

```
hsocket:DISCONNECT().
DELETE OBJECT hsocket.
```

11

XML Support

Extensible Markup Language (XML) is a data format for structured document interchange over networks. This chapter describes language extensions to the Progress 4GL that enable Progress applications to use XML through the integration of a third-party DOM parser.

This chapter includes the following sections:

NOTE: For complete information on using the Simple API for XML (SAX) with the Progress 4GL, see Chapter 12, “Simple API For XML (SAX),” in this manual.

11.1 Introduction

The Extensible Markup Language (XML) is a data format for structured document interchange on the Web and other networks. It is hardware architecture neutral and application independent. XML documents are composed of storage units called *entities*, that contain either parsed or unparsed data. Parsed data is made up of characters, some of which form character data, and some of which form *markup*. Markup encodes a description of a document's storage layout and logical structure.

A software module called an XML processor is used to read XML documents and provide access to their content and structure. It is assumed that an XML processor is doing its work on behalf of another module call the application.

11.1.1 XML Documents

XML documents are made up of two parts: a prologue and a body.

- The *prologue* contains optional information such as the XML version the document conforms to, information about the character encoding used to encode the contents of the document, and a *document type definition* (DTD) which describes the grammar and vocabulary of the document.
- The *body* may contain elements, entity references, and other markup information.

DTDs are rules that define the elements that can exist in a particular document or group of documents, and the relationships among the various elements. A DTD can be part of the content of an XML document or can be separate from it and referred to by the documents. Here is an example of a DTD:

```
<? xml version="1.0" encoding="UTF-8" ?>
```

Elements represent the logical components of documents. They can contain data or other elements. For example, a customer element can contain a number of column (field) elements and each column element one data value. Here is an example of an element:

```
<name>Clyde</name>
```

Elements can have additional information called *attributes* attached to them. Attributes describe properties of elements. Here is an example of an element with an attribute, `emp-num`:

```
<name emp-num="1">Mary</name>
```

Here is an example of elements that contain other elements:

```
<phone><entry><name>Chip</name><extension>3</extension></entry></phone>
```

11.2 The Document Object Model

The Document Object Model (DOM) is an application programming interface (API) for XML documents. It defines the logical structure of documents and the way a document is accessed and manipulated. In the DOM specification, the term "document" is used in the broad sense to include many different kinds of information that might be stored in diverse systems. Much of this would traditionally be seen as data rather than as documents. Nevertheless, XML presents this data as documents, and the DOM manages this data.

When you read an XML document via the DOM API, the DOM parser reads and parses the complete input document before making it available to the application.

Progress has defined an initial set of extensions to the Progress 4GL to allow the use of XML through the DOM interface. These extensions provide 4GL applications with the *basic* input, output, and low-level data manipulation capabilities required to use data contained in XML documents. They are not intended to provide access to the entire DOM interface, nor are they intended to include all the high-level constructs.

Note On DOM Compatibility With the 4GL

The DOM API is designed to be compatible with a wide range of programming languages, but the naming convention chosen by the World Wide Web Consortium (W3C) does not match what already exists in the Progress 4GL. In some cases, PSC elected to use the familiar names already used in the 4GL rather than the names given in the DOM specification. Similarly, where there are existing 4GL features that provide the same capability as the DOM interfaces, PSC has chosen to use the 4GL implementation rather than introduce new language features that match the DOM more closely.

11.2.1 The DOM Structure Model

The DOM presents documents as a hierarchy or tree of *node* objects that also implement other, more specialized interfaces. Some types of nodes may have *child nodes* of various types, and others are *leaf nodes* that cannot have anything below them in the document structure. The node types, and which node types they may have as children, are shown in Table 11–1.

Table 11–1: Node Interface Types *(1 of 2)*

Node Interfaces	Description	Children
DocumentType	Represents the Document Type Definition or Schema declaration of the XML document.	Notation, Entity
DocumentFragment	Represents a lightweight object used to store sections of an XML document temporarily.	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
EntityReference	Represents a reference to an entity within the XML document.	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
Element	Represents an element node. This interface represents the data, or the tags of the XML document. The text of the element is stored in a Text or CDATASection node, which is the child of the element.	Element, Text, Comment, ProcessingInstruction, CDATASection, EntityReference
Attribute	Represents an attribute of a document or an element. The allowable values for the attribute are defined in a document type definition. Attributes are NOT considered as child nodes of the element they describe.	Text, EntityReference
CDATASection	CDATA sections are used to escape blocks of text that would otherwise be regarded as markup. The primary purpose is for including XML fragments, without needing to escape all the delimiters.	None
Comment	Represents the content of a comment	None
Entity	Represents an entity, either parsed or unparsed, in the XML document.	None

Table 11–1: Node Interface Types *(2 of 2)*

Node Interfaces	Description	Children
Notation	Represents a notation declared within the DTD.	None
ProcessingInstruction	The “Processing Instruction” is a way to keep processor-specific information in the text of the document	None
Text	Represents a Text node that is a child of an element node	None

11.3 The New 4GL Objects and Handles

11.3.1 DOM Node Interfaces As Subtypes

Since it is necessary to be able to use some of the other specialized interfaces as well as the simplified interface, the DOM objects that have these specialized interfaces are implemented as Subtypes of the Progress object.

The Progress 4GL supports the following interfaces as Subtypes on the X-NODEREF:

- CDATA-SECTION
- COMMENT
- DOCUMENT-FRAGMENT
- ELEMENT
- ENTITY-REFERENCE
- PROCESSING-INSTRUCTION
- TEXT

The default Subtype will be ELEMENT.

Table 11–2 shows the “simplified” DOM Node interface nodeNames and nodeValues.

Table 11–2: Node Names and Values

Type	nodeName	nodeValue
DocumentType	Document type name	Null
DocumentFragment	#document-fragment	Null
Element	Tag name	Null
EntityReference	Name of entity referenced	Null
Text	#text	Content of the text node
CDATASection	#cdata-section	Content of the CDATA-section
Comment	#comment	Content of the comment
ProcessingInstruction	Target	Content excluding the target

The Progress 4GL NAME attribute will be used to return the nodeName, while a new NODE-VALUE character attribute will return or set the node’s nodeValue.

11.3.2 Document and Node Reference Objects

The X-DOCUMENT is a Progress 4GL object that represents a DOM Document object. The X-DOCUMENT object is assigned to the HANDLE data type and is used to manipulate the XML document and its tree representation.

The X-NODEREF object is a Progress 4GL object that is a reference to any node in an XML tree except a Document node. The X-NODEREF object is assigned to the HANDLE data type and is used to manipulate the DOM nodes. Note that an X-NODEREF object is not an actual node in the XML tree but is more like a cursor which is used to navigate the tree and manipulate the nodes in it.

Creating a Document Object

The creation and saving of a document is not part of the DOM Core API, but is left to the application that calls the API. You create an XML document using the X-DOCUMENT option of the CREATE Widget statement:

```
DEFINE VARIABLE hXDoc AS HANDLE.
CREATE X-DOCUMENT hXDoc.
```

This statement creates a handle for an object of the type “X-DOCUMENT” that “wraps” an XML document. You may start adding nodes to it or use the LOAD( ) method to populate it from an existing XML document.

Creating a Node Reference Object

You add a node reference object to the XML document using the X-NODEREF option of the CREATE Widget statement:

```
DEFINE VARIABLE hNRef AS HANDLE.
CREATE X-NODEREF hNRef.
```

This statement creates a handle for an object which is not an actual node, but which is used as a reference or pointer to an actual node in the tree. The X-NODEREF object provides a path to access and manipulate the actual document nodes and can be used as a parameter or as a return-value for methods that will associate the handle with an XML node.

11.4 Creating XML Output From the 4GL

In order to create an output XML document, you must complete the following steps:

1 ♦ Create an X-DOCUMENT object.

2 ♦ Create a Root Node reference object.

3 ♦ Create a Node reference object for each type of node.

4 ♦ Create the root node and append it to the document.

5 ♦ Create each specific node required.

6 ♦ Append each node to its parent.

7 ♦ Set node attributes.

8 ♦ Steps 5 through 7 are iterative.

9 ♦ Save the document as an XML file.

10 ♦ Delete the objects.

11.4.1 The Root Node Reference Object

The *root node* is the unique top-level node that is not a child of any other node. All other nodes are children or other descendents of the root node. A root node is necessary so that you have a top-level node to which you can append the child nodes.

11.4.2 Creating and Appending a Node

In order to create an actual XML node, you use the CREATE-NODE( ) method on the parent object. After the node is created, you must append it to its parent by using the APPEND-CHILD( ) method. The following code fragment is an example of creating and appending the root node:

```
CREATE X-NODEREF hRoot.
hDoc:CREATE-NODE(hRoot,"Root","ELEMENT").
hDoc:APPEND-CHILD(hRoot).
. . .
```

11.4.3 Setting Node Attributes and Values

You can set the attributes of a node or the value of a node either before or after it is appended by using the SET-ATTRIBUTE( ) method or the NODE-VALUE attribute. The following code fragment depicts setting attributes of the “employee” ELEMENT node with the SET-ATTRIBUTE( ) method and setting the value of the “address” TEXT node with the NODE-VALUE attribute. Note that in this case, the “employee” node is a child of the root node and the “address” node is a child of the “employee” node:

```
hDoc:CREATE-NODE(hEmp,"employee","ELEMENT").
hDoc:CREATE-NODE(hAddr,?,"TEXT").

hEmp:SET-ATTRIBUTE("empID","10263").
hEmp:SET-ATTRIBUTE("empDept","Sales").
hRoot:APPEND-CHILD(hEmp).

hEmp:APPEND-CHILD(hAddr).
hAddr:NODE-VALUE = "121 State Street".
. . .
```

For more information on attributes and methods associated with the X-DOCUMENT object and the X-NODEREF object, see their entries in the *Progress Language Reference* manual.

11.4.4 Example Of Creating an Output XML File

The following sample program creates an XML file consisting of all fields in all the customer records where the cust-num is less than “5". You must use the SAVE( ) method on the X-DOCUMENT object in order to create the actual XML file:

e-outcus.p *(1 of 2)*

```
/* e-outcus.p - Export the Customer table to an xml file*/
DEFINE VARIABLE hDoc AS HANDLE.
DEFINE VARIABLE hRoot AS HANDLE.
DEFINE VARIABLE hRow AS HANDLE.
DEFINE VARIABLE hField AS HANDLE.
DEFINE VARIABLE hText AS HANDLE.
DEFINE VARIABLE hBuf AS HANDLE.
DEFINE VARIABLE hDBFld AS HANDLE.
DEFINE VARIABLE i AS INTEGER.

CREATE X-DOCUMENT hDoc.
CREATE X-NODEREF hRoot.
CREATE X-NODEREF hRow.
CREATE X-NODEREF hField.
CREATE X-NODEREF hText.

hBuf = BUFFER customer:HANDLE.

/*set up a root node*/
hDoc:CREATE-NODE(hRoot,"Customers","ELEMENT").
hDoc:APPEND-CHILD(hRoot).
FOR EACH customer WHERE cust-num < 5:
    hDoc:CREATE-NODE(hRow,"Customer","ELEMENT"). /*create a row node*/
    hRoot:APPEND-CHILD(hRow).  /*put the row in the tree*/
    hRow:SET-ATTRIBUTE("Cust-num",STRING(cust-num)).
    hRow:SET-ATTRIBUTE("Name",NAME).
    /*Add the other fields as tags in the xml*/
    REPEAT i = 1 TO hBuf:NUM-FIELDS:
        hDBFld = hBuf:BUFFER-FIELD(i).
        IF hDBFld:NAME = "Cust-num" OR  hDBFld:NAME = "NAME" THEN NEXT.
        /*create a tag with the field name*/
        hDoc:CREATE-NODE(hField, hDBFld:NAME, "ELEMENT").
        /*put the new field as next child of row*/
        hRow:APPEND-CHILD(hField).
        /*add a node to hold field value*/
        hDoc:CREATE-NODE(hText, "", "TEXT").
```

e-outcus.p *(2 of 2)*

```
        /*attach the text to the field*/
        hField:APPEND-CHILD(hText).
        hText:NODE-VALUE = STRING(hDBFld:BUFFER-VALUE).
    END.
END.
/*write the XML node tree to an xml file*/
hDoc:SAVE("file","cust.xml").

DELETE OBJECT hDoc.
DELETE OBJECT hRoot.
DELETE OBJECT hRow.
DELETE OBJECT hField.
DELETE OBJECT hText.
```

A partial output of the above program appears below. Note that the carriage returns and indentations have been entered for readability; the actual file contains one long string:

```
<?xml version='1.0' ?>
<Customers>
  <Customer Name="Lift Line Skiing" Cust-num="1">
    <Country>USA</Country>
    <Address>276 North Street</Address>
    <Address2></Address2>
    <City>Boston</City>
    <State>MA</State>
    <Postal-Code>02114</Postal-Code>
    <Contact>Gloria Shepley</Contact>
    <Phone>(617) 450-0087</Phone>
    <Sales-Rep>HXM</Sales-Rep>
    <Credit-Limit>66700</Credit-Limit>
    <Balance>42568</Balance>
    <Terms>Net30</Terms>
    <Discount>35</Discount>
    <Comments>This customer is on credit hold.</Comments>
  </Customer>
  <Customer Name="Urpon Frisbee" Cust-num="2">
    <Country>Finland</Country>
    <Address>Rattipolku 3</Address>
     . . .

  </Customer>
</Customers>
```

Writing an XML File To a MEMPTR Or a Stream

You can also write an XML file to a MEMPTR or to an output stream as the following code fragments demonstrate. The following fragment shows saving an XML file to a MEMPTR:

```
DEFINE VARIABLE memfile AS MEMPTR.
. . .
hDoc:SAVE("memptr",memfile). /* SAVE() will set the memptr size */
. . .
```

The following fragment displays saving an XML file to an output stream:

```
DEFINE STREAM xmlstream.
. . .
OUTPUT STREAM xmlstream TO custxml.xml.
hDoc:SAVE("stream","xmlstream").
OUTPUT CLOSE.
. . .
```

11.5 Reading XML Input Into the 4GL

In order to read in an XML file and process it, you must complete the following steps:

1 ♦ Create an X-DOCUMENT object.

2 ♦ Create a root node reference object.

3 ♦ Use the LOAD( ) method to read the input file.

4 ♦ Use the GET-DOCUMENT-ELEMENT( ) method to get the root node reference handle.

5 ♦ Create a node reference object.

6 ♦ Using the GET-CHILD( ) method, read through the child nodes.

7 ♦ Using the GET-ATTRIBUTE( ) METHOD, the NAME attribute, the NODE-VALUE attribute and other attributes and methods, access the XML data.

8 ♦ Update the database or other fields as necessary.

9 ♦ Delete the objects.

11.5.1 Loading an XML File

The LOAD( ) method reads the specified file into memory, parses it, optionally validates it, and makes its contents available to the 4GL. Once the XML file is in memory, you must get the handle to its root element by using the GET-DOCUMENT-ELEMENT( ) method. Once you have the root node handle, you can manipulate the remaining child nodes.

The following code fragment demonstrates loading an XML file called “myfile.xml”:

```
DEFINE VARIABLE hDocMine AS HANDLE.
DEFINE VARIABLE hRootMine AS HANDLE.

CREATE X-DOCUMENT hDocMine.
CREATE X-NODEREF hRootMine.

hDocMine:LOAD("FILE","myfile.xml",TRUE).
hDocMine:GET-DOCUMENT-ELEMENT(hRootMine).
. . .
```

Loading an XML File From a MEMPTR

An XML file can be read from a MEMPTR as the following code fragment demonstrates:

```
DEFINE VARIABLE memfile AS MEMPTR.
. . .
FILE-INFO:FILE-NAME = "meminp.xml".
SET-SIZE(memfile) = FILE-INFO:FILE-SIZE.
INPUT FROM "meminp.xml" BINARY NO-CONVERT.
hDoc:LOAD("memptr",memfile,FALSE).
. . .
```

11.5.2 Accessing the Child Nodes

Before you can work with a child node of an XML document, you must create a node reference object with which to access the node. Then you access the node by using the GET-CHILD( ) method. The following code fragment shows obtaining the third child node of the parent node, hParent:

```
. . .
DEFINE VARIABLE hChildNode AS HANDLE.
CREATE X-NODEREF hChildNode.
logvar = hParent:GET-CHILD(hChildNode,3).
. . .
```

11.5.3 Using Node Attributes and Values

You can get information about the child node by using various attributes and methods. For example, if you do not know how many nodes there are below the node referred to by the node reference, you can use the NUM-CHILDREN attribute. You can obtain or set the value of the node by using the NODE-VALUE attribute:

```
DEFINE VARIABLE hChNode AS HANDLE.
CREATE X-NODEREF hChNode.
REPEAT i = 1 TO hParent:NUM-CHILDREN:
    logvar = hParent:GET-CHILD(hChNode, i).
    IF hChNode:NODE-VALUE > 0 THEN
        hChNode:NODE-VALUE = hChNode:NODE-VALUE + i.
. . .
```

You can obtain a list of an element's attribute names using the ATTRIBUTE-NAMES attribute, get the value of an attribute by using the GET-ATTRIBUTE( ) method or set the value of an attribute by using the SET-ATTRIBUTE( ) method. You can also REMOVE-ATTRIBUTE( ):

```
. . .
REPEAT i = 1 TO hNode1:NUM-CHILDREN:
  logvar = hNode1:GET-CHILD(hChNode, i).
  IF NOT logvar THEN LEAVE.
  entries = hNode1:ATTRIBUTE-NAMES.
  REPEAT j = 1 TO NUM-ENTRIES(entries):
    aname = ENTRY(j, entries).
    MESSAGE "attrname is " aname "value is " hNode1:GET-ATTRIBUTE(aname).
  END.
END.
. . .
```

In addition to creating nodes, you can IMPORT-NODE( ), CLONE-NODE( ), and DELETE-NODE( ). In addition to appending and getting a child, you can REMOVE-CHILD(), REPLACE-CHILD( ), and GET-PARENT( ). The following example demonstrates the CLONE-NODE( ) method:

e-clone.p *(1 of 2)*

```
/* e-clone.p */
DEFINE VARIABLE hXref AS HANDLE.
DEFINE VARIABLE hXref1 AS HANDLE.
DEFINE VARIABLE hText AS HANDLE.
DEFINE VARIABLE hText1 AS HANDLE.
DEFINE VARIABLE hClone AS HANDLE.
DEFINE VARIABLE hRoot AS HANDLE.
DEFINE VARIABLE hDoc AS HANDLE.

CREATE X-NODEREF hXref.
CREATE X-NODEREF hXref1.
CREATE X-NODEREF hText.
CREATE X-NODEREF hText1.
CREATE X-NODEREF hClone.
CREATE X-NODEREF hRoot.
CREATE X-DOCUMENT hDoc.

hDoc:CREATE-NODE(hRoot,"root","ELEMENT").
hDoc:INSERT-BEFORE(hRoot,?).
hDoc:CREATE-NODE(hXref,"customer","ELEMENT").
hDoc:CREATE-NODE(hXref1,"order","ELEMENT").
hDoc:CREATE-NODE(hText,?,"TEXT").
hDoc:CREATE-NODE(hText1,?,"TEXT").

/* Add the two element nodes to the root, each with a text*/
hXref:SET-ATTRIBUTE("id","54").
hXref:SET-ATTRIBUTE("name","Second Skin Scuba").
hRoot:APPEND-CHILD(hXref).
hXref:APPEND-CHILD(hText).

hXref1:SET-ATTRIBUTE("id","55").
hXref1:SET-ATTRIBUTE("name","Off the Wall").
hRoot:APPEND-CHILD(hXref1).
hXref1:APPEND-CHILD(hText1).
hText:NODE-VALUE = "hi from customer".
hText1:NODE-VALUE = "hi from order".

hXref1:CLONE-NODE(hClone,TRUE).
hRoot:APPEND-CHILD(hClone).
```

e-clone.p *(2 of 2)*

```
/* Save the file */
hDoc:SAVE("file","clone1.xml").
DELETE OBJECT hXref.
DELETE OBJECT hXref1.
DELETE OBJECT hText.
DELETE OBJECT hText1.
DELETE OBJECT hClone.
DELETE OBJECT hRoot.
DELETE OBJECT hDoc.
```

There are more methods and attributes that apply to the X-DOCUMENT object and the X-NODEREF objects. For more information on these attributes and methods, see their entries in the *Progress Language Reference* manual.

11.5.4 Examples Of Reading an Input XML File

The following sample program shows reading in a file called “personal.xml”, processing through all the child nodes and displaying information if the node name is “person”:

e-attnam.p

```
/* e-attnam.p */
DEFINE VARIABLE hDoc AS HANDLE.
DEFINE VARIABLE hRoot AS HANDLE.
DEFINE VARIABLE good AS LOGICAL.

CREATE X-DOCUMENT hDoc.
CREATE X-NODEREF hRoot.

hDoc:LOAD("file","personal.xml",TRUE).
hDoc:GET-DOCUMENT-ELEMENT(hRoot).

RUN GetChildren(hRoot, 1).
DELETE OBJECT hDoc.
DELETE OBJECT hRoot.

PROCEDURE GetChildren:
DEFINE INPUT PARAMETER hParent AS HANDLE.
DEFINE INPUT PARAMETER level AS INTEGER.
DEFINE VARIABLE i AS INTEGER.
DEFINE VARIABLE hNoderef AS HANDLE.

CREATE X-NODEREF hNoderef.

REPEAT i = 1 TO hParent:NUM-CHILDREN:
    good = hParent:GET-CHILD(hNoderef,i).
    IF NOT good THEN LEAVE.
    IF hNoderef:SUBTYPE <> "element" THEN NEXT.
    IF hNoderef:NAME = "person" THEN
        MESSAGE "getattr id gives" hNoderef:GET-ATTRIBUTE("id")
          hNoderef:ATTRIBUTE-NAMES.
    RUN GetChildren(hNoderef, (level + 1)).
END.

DELETE OBJECT hNoderef.
END PROCEDURE.
```

The following program reads in the output file created by the previous program, e-outcus.p and creates temp-table entries:

e-incus.p *(1 of 2)*

```
/* e-incus.p - Import the Customer table from an xml file*/
DEFINE VARIABLE hDoc AS HANDLE.
DEFINE VARIABLE hRoot AS HANDLE.
DEFINE VARIABLE hTable AS HANDLE.
DEFINE VARIABLE hField AS HANDLE.
DEFINE VARIABLE hText AS HANDLE.
DEFINE VARIABLE hBuf AS HANDLE.
DEFINE VARIABLE hDBFld AS HANDLE.
DEFINE VARIABLE i AS INTEGER.
DEFINE VARIABLE j AS INTEGER.

/*so we can create new recs*/
DEFINE TEMP-TABLE Custt LIKE Customer.

CREATE X-DOCUMENT hDoc.
CREATE X-NODEREF hRoot.
CREATE X-NODEREF hTable.
CREATE X-NODEREF hField.
CREATE X-NODEREF hText.
hBuf = BUFFER Custt:HANDLE.

/*read in the file created in the last example*/
hDoc:LOAD("file", "cust.xml", FALSE).
hDoc:GET-DOCUMENT-ELEMENT(hRoot).

/*read each Customer from the root*/
REPEAT i = 1 TO hRoot:NUM-CHILDREN:
    hRoot:GET-CHILD(hTable,i).
    CREATE Custt.
    /*get the fields given as attributes*/
    cust-num = integer(hTable:GET-ATTRIBUTE("Cust-num")).
    NAME = hTable:GET-ATTRIBUTE("Name").
    /*get the remaining fields given as elements with text*/
    REPEAT j = 1 TO hTable:NUM-CHILDREN:
       hTable:GET-CHILD(hField,j).
       IF hField:NUM-CHILDREN < 1 THEN NEXT.
    /*skip any null value*/
       hDBFld = hBuf:BUFFER-FIELD(hField:NAME).
       hField:GET-CHILD(hText,1).
 /*get the text value of the field*/
       hDBFld:BUFFER-VALUE = hTEXT:NODE-VALUE.
    END.
END.
```

e-incus.p *(2 of 2)*

```
DELETE OBJECT hDoc.
DELETE OBJECT hRoot.
DELETE OBJECT hTable.
DELETE OBJECT hField.
DELETE OBJECT hText.

/* show data made it by displaying temp-table */
FOR EACH Custt:
    DISPLAY custt.
END.
```

11.6 Internationalization

XML documents may be encoded using any of a wide a variety of character encoding. The DOM parser returns character data to the Progress 4GL interpreter encoded, if possible, according to -cpinternal, the Internal Code Page parameter. This translation is performed by the DOM parser using its own translation functions. If the DOM parser cannot do the translation according to -cpinternal, it translates to UTF8 which is then translated by the interpreter from UNICODE to the character encoding specified by -cpinternal. The encoding used in an XML document is specified by an optional *encoding declaration* at its very beginning. If the encoding declaration is present, it specifies the encoding used in the remainder of the document. If the declaration is not present, the document's encoding is assumed to be UTF-8 or UTF-16.

When the LOAD method is used to load an XML document, the ENCODING attribute of the X-DOCUMENT will be set to the name of encoding found in the encoding declaration of the document. For output, you can set the X-DOCUMENT's ENCODING attribute to the name of the desired encoding.

When the SAVE method is used to write an output XML document from a memory-resident DOM tree, the generated XML text is encoded by the DOM parser according to the value of the ENCODING attribute. When you SAVE a document to a stream, the specified encoding is used and the value of -cpstream is ignored.

According to the XML recommendation, "it is a fatal error when an XML processor encounters an entity with an encoding that it is unable to process". If this error occurs while Progress is attempting to load a document, the document will be empty.

11.7 Error Handling

Any of the methods listed above may fail, but this will not normally cause the Progress error status to be raised. Instead, the method will return FALSE if that is appropriate. Also, the parsing may encounter errors that do not cause the operation as a whole to fail. So instead of testing for ERROR-STATUS:ERROR after running a method with NO-ERROR, you should test for ERROR-STATUS:NUM-MESSAGES being greater than 0.

Note that the DOM parser may detect errors in an input XML document even if validation is not specified in the LOAD( ) method call. Validation checks the document for conformance to a DTD, but there could be other errors, such a missing end-tag or mismatched tags. The parser will report these errors independently of validation against a DTD.

12

Simple API For XML (SAX)

This chapter tells you how to use the Simple API for XML, commonly known as SAX, with the Progress 4GL.

This chapter assumes that the reader is an experienced Progress developer who understands XML, SAX, Progress AppBuilder (if the reader wants to develop SAX applications using the Progress Application Development Environment (ADE)) and Progress WebSpeed (if the reader wants to use SAX in Progress WebSpeed applications).

For more information on XML, see the XML 1.0 specification, available at `http://www.w3.org`. For more information on SAX, see the SAX 2.0 specification, available at `http://www.saxproject.org/apidoc`. For more information on the Progress ADE, see the *Progress AppBuilder Developer's Guide*. For more information on Progress WebSpeed, see the *WebSpeed Installation and Configuration Guide* and the *WebSpeed Developer's Guide*.

This chapter contains the followings sections:

12.1 About SAX

The Simple API for XML (SAX) is an application programming interface (API) for XML documents. It defines the way an XML document is accessed and manipulated. In the SAX specification, the term “document” is used in the broad sense to include many different kinds of information that might be stored in diverse systems. Much of this would traditionally be seen as data rather than as documents. Nevertheless, XML presents this data as documents, and SAX lets you manage them.

When an XML document is accessed by a SAX application, as the XML parser encounters an XML element, it parses that element and provides its information to the application immediately, through a callback (explained in the “Progress SAX Callbacks” section and defined in the “SAX Callback Reference” section of this book). This contrasts with the Document Object Model (DOM), which requires the entire XML document to be loaded before information on the document’s XML elements is provided.

Note On SAX Compatibility With the Progress 4GL

The SAX API is designed to be compatible with a wide range of programming languages. For the most part, Progress has followed the SAX Java API naming conventions. In a very few cases, Progress Software Corporation (PSC) elected to use the familiar names already used in the 4GL rather than the names given in the SAX specification. Similarly, where there are existing 4GL features that provide the same capability as the SAX interfaces, PSC has chosen to use the 4GL implementation rather than introduce new language features that match SAX more closely.

12.2 Why Use Progress SAX?

The advantages of SAX over the Document Object Model (DOM) interface are well known. Two major advantages are:

1. SAX requires much less memory than DOM does.

2. An application using SAX can stop the XML parser as soon as it has all the data it needs, even if it has parsed only part of the XML document. This might save a lot of processing time.

Yet, Progress SAX is just as easy to set up as Progress DOM.

12.3 Understanding Progress SAX

If you have already used Progress DOM, you know it provides the 4GL objects X-DOCUMENT and X-NODEREF and their attributes and methods, all of which are documented in the *Progress Language Reference*.

Similarly, Progress SAX interface provides two 4GL objects, SAX-READER and SAX-ATTRIBUTES, and their attributes and methods.

In addition, Progress SAX lets you do the following: provide callbacks for it to invoke, toggle validation and namespace processing, parse an XML document with one call or multiple calls, monitor the state of the parse, and detect and handle errors.

These topics are discussed in the following sections:

- SAX-READER Object
- Progress SAX Callbacks
- SAX-ATTRIBUTES Object
- Validation
- Namespace Processing
- Parsing With One Call Or With Multiple Calls
- Monitoring the State Of the Parse
- Error Handling

12.3.1 SAX-READER Object

The application creates a SAX-READER object and uses it to control the XML parser.

The application can parse an XML document in one call or in multiple calls. For more information on this, see the "Parsing With One Call Or With Multiple Calls" section in this document.

When the application is finished parsing the XML document, Progress Software Corporation recommends that the application deletes the SAX-READER object.

The attributes and methods associated with the SAX-READER object are summarized in Table 12–1. For reference entries, see the *Progress Version 9 Product Update Bulletin*.

Table 12–1: SAX-READER Attribute and Method Summary

This Attribute Or Method...	Lets You...
SET-INPUT-SOURCE() Method	Specify the XML input
SCHEMA-PATH Attribute	Specify a search path for the Document Type Definition (DTD) or any other external entities
HANDLER Attribute	Tell the parser where the callbacks reside
SUPPRESS-NAMESPACE-PROCESSING Attribute VALIDATION-ENABLED Attribute	Toggle parser options
SAX-PARSE() Method SAX-PARSE-FIRST() Method SAX-PARSE-NEXT() Method STOP-PARSING() Method	Start, continue, or stop parsing
LOCATOR-COLUMN-NUMBER Attribute LOCATOR-LINE-NUMBER Attribute LOCATOR-PUBLIC-ID Attribute LOCATOR-SYSTEM-ID Attribute PARSE-STATUS Attribute	Get the status of the parse
TYPE Attribute	Get the type of the object (which is always “SAX-READER”)
ADM-DATA Attribute PRIVATE-DATA Attribute UNIQUE-ID Attribute	Get or set information concerning this particular SAX-READER object

12.3.2 Progress SAX Callbacks

When the XML parser encounters an XML token, Progress invokes the callback corresponding to that token — if that callback is provided by the developer. If not, the parser continues.

Progress SAX implements callbacks as internal procedures coded by the 4GL SAX developer using signatures specified by Progress. Normally, callbacks are placed in a procedure (`.p`) file that the application runs persistently. But callbacks can also be placed in the driver routine, which is the routine that calls the SAX-PARSE() or SAX-PARSE-FIRST() method. In whichever procedure callbacks are placed, the application assigns that procedure's handle to the SAX-READER's HANDLER attribute. (HANDLER defaults to a handle to the driver routine). Then, the application starts the parse.

NOTE: Although there are many callbacks in the SAX2 interface, your application might need to use only a few — most likely StartElement, Characters, and EndElement. You might not care about the other callbacks.

Within a callback, to get a handle to the SAX-READER object that invoked the callback, use the SELF system handle.

The following fragment uses SELF within a callback to call the SAX-READER's STOP-PARSING() method:

```
SELF:STOP-PARSING().
```

The following fragment uses SELF within a callback to store data in the SAX-READER's PRIVATE-DATA attribute:

```
SELF:PRIVATE-DATA = "xyz123".
```

For information on the SAX parser's current location in the XML source, use the following attributes of SAX-READER:

- LOCATOR-COLUMN-NUMBER
- LOCATOR-LINE-NUMBER
- LOCATOR-PUBLIC-ID
- LOCATOR-SYSTEM-ID

NOTE: These attributes are valid only within a callback.

Table 12–2 summarizes the callbacks. For complete descriptions, see the "SAX Callback Reference" section in this chapter.

Table 12–2: SAX Callback Summary

This Callback...	Lets You...
ResolveEntity	Tell the parser where to find an external entity
StartDocument ProcessingInstruction StartPrefixMapping EndPrefixMapping StartElement Characters IgnorableWhitespace EndElement EndDocument	Process various XML tokens
NotationDecl UnparsedEntityDecl	Process notations and unparsed entities
Warning Error FatalError	Handle errors

12.3.3 SAX-ATTRIBUTES Object

When the parser encounters an XML Start-Tag that has one or more attributes, such as:

```
<Entry name="John Smith" id="2543">
```

it populates the Start Element callback's *attributes* parameter with information on the element's attributes (in the example, "name" and "id"). In Progress, the *attributes* parameter is a handle to a Progress SAX-ATTRIBUTES object, which is similar to the Attributes interface of the Java Sax2 API.

All SAX-ATTRIBUTES objects are created and destroyed by Progress. The application cannot create a SAX-ATTRIBUTES object. The application can, however, store data residing in the SAX-ATTRIBUTES object in data items under the control of the application.

The attributes and methods associated with the SAX-ATTRIBUTES object are summarized in Table 12–3. For reference entries, see the *Progress Version 9 Product Update Bulletin*.

Table 12–3: SAX-ATTRIBUTES Attribute and Method Summary *(1 of 2)*

This Attribute or Method...	Lets you...
NUM-ITEMS Attribute	Get how many attributes the XML element has
GET-INDEX-BY-NAMESPACE-NAME() Method GET-INDEX-BY-QNAME() Method	Get where on the attribute list a particular attribute resides
GET-LOCALNAME-BY-INDEX() Method GET-QNAME-BY-INDEX() Method GET-URI-BY-INDEX() Method	Get the name of a particular attribute
GET-TYPE-BY-INDEX() Method GET-TYPE-BY-NAMESPACE-NAME() Method GET-TYPE-BY-QNAME Method	Get the type of a particular attribute

Table 12–3: SAX-ATTRIBUTES Attribute and Method Summary *(2 of 2)*

This Attribute or Method...	Lets you...
GET-VALUE-BY-INDEX() Method GET-VALUE-BY-NAMESPACE-NAME() Method GET-VALUE-BY-QNAME() Method	Get the value of a particular attribute
TYPE Attribute	Get the type of the object (which is always “SAX-ATTRIBUTE”)
ADM-DATA Attribute PRIVATE-DATA Attribute UNIQUE-ID Attribute	Get or set information concerning this particular SAX-ATTRIBUTES object

A SAX-ATTRIBUTES object is a collection of all the attributes for a given element. No matter how many attributes an element has, StartElement gets passed only one attributes handle. Think of it as a list of the attributes associated with the element,

NOTE: The order of the elements on the list might not be the same as the order in which they appear in the document — which is consistent with the SAX2 Java API specification.

To get the information about each attribute from a SAX-ATTRIBUTES object, use the GET-XXX methods. They let your application to get the SAX-ATTRIBUTES data in either of two different ways:

- You can traverse the list, getting each attribute’s data in turn by using the GET-XXX-BY-INDEX methods.
- You can use an attribute’s name to get its data.

  This approach has variations, depending on whether you are using Namespace processing or not. For more information on this point, see the “Namespace Processing” section of this chapter.)

There are 6 pieces of information you can get on each attribute, though three are variations on the name. Each GET-XXX method gets you one piece of information. The six pieces of information and the GET-XXX methods are explained in Table 12–4.

Table 12–4: The GET-XXX Methods

To Get This Information On an Attribute...	Use These Methods...
Its position on the list	GET-INDEX-BY-NAMESPACE-NAME() GET-INDEX-BY-QNAME()
Its Namespace URI, if namespace information is available	GET-URI-BY-INDEX()
Its localName, if namespace processing is being enabled	GET-LOCALNAME-BY-INDEX()
Its Qualified name (qName)	GET-QNAME-BY-INDEX()
Its XML attribute type (declared in the DTD)	GET-TYPE-BY-INDEX() GET-TYPE-BY-NAMESPACE-NAME() GET-TYPE-BY-QNAME()
Its value	GET-VALUE-BY-INDEX() GET-VALUE-BY-NAMESPACE-NAME() GET-VALUE-BY-QNAME()

If the parser is doing Namespace processing — that is, if SAX-READER's SUPPRESS-NAMESPACE-PROCESSING attribute is set to "NO" (the default) — each attribute that has a namespace prefix will have non-empty URI, localName, and qName data. An attribute that has noNamespace prefix will have an empty URI, but its localName and qName will have values.

NOTE: An unprefixed attribute name does **not** use the default Namespace, if any; it is not associated with any Namespace. This contrasts with the case for Elements, where unprefixed Element names use the default Namespace, if any.

If the parser is not doing Namespace processing, each attribute will have only a qName.

In all cases, the qName will be exactly what appears in the XML document.

12.3.4 Validation

Progress SAX always checks that the XML document is well formed. In addition, Progress SAX can validate the XML document against a DTD.

A DTD might be completely specified in the XML source. Alternatively, part of all of the DTD might reside in one or more external files named in the XML source. In the latter case, to override the locations given in the XML source, use one or both of the following techniques:

- Implement the ResolveEntity callback
- Set the SAX-READER's SCHEMA-PATH attribute

If you do not want Progress SAX to validate the XML document against a DTD, set the VALIDATION-ENABLED attribute to FALSE.

NOTE: Even when VALIDATION-ENABLED is NO, the parser still reads any specified DTD (internal or external) in order to get information on entities.

For more information, see the reference entries for the VALIDATION-ENABLED and SCHEMA-PATH attributes in the *Progress Version 9 Product Update Bulletin*.

12.3.5 Namespace Processing

Namespace processing is enabled by default.

To disable it, set SUPPRESS-NAMESPACE-PROCESSING to TRUE.

XML Namespace processing is potentially complex. But here is a summary of the behavior to expect from a Progress SAX application.

Namespace Declarations

An XML document that uses namespaces has one or more Namespace declarations, which appear as attributes of elements. A namespace declaration might appear as an attribute of any element; frequently the document element (the one that encloses all of the other elements) has the namespace declaration or declarations.

A namespace declaration associates a prefix with a URI. Once associated, the prefix might appear with element names, attributes names, or both, to distinguish the names from identical names that might mean something else. For example, an XML element whose name is “memory” might mean completely different things, and have different valid attributes, depending on whether it appears in a computer inventory or in a psychological report. You can distinguish the uses by having a computer-inventory namespace and a psychological-report namespace.

In the SAX2 interface, XML Namespaces affect the values of the parameters of StartElement and EndElement, and the attribute data in the attributes parameter of StartElement. There can be slight variations in the way that Progress SAX handles namespace processing.

Namespace Processing's Effects

In Progress SAX, Namespace processing affects the behavior of:

- The StartElement and EndElement callbacks
- Attributes data

Table 12–5 describes the effect of Namespace processing on StartElement and EndElement.

Table 12–5: Effect Of Namespace Processing On StartElement and EndElement

If Namespace Processing Is Enabled...	If Namespace Processing Is Suppressed...
If an element's name has a namespace prefix: • namespaceURI will be the URI associated with the prefix • localName will be the name as given in the XML doc, without the prefix • qName will be the name as given in the XML doc If an element's name has no prefix: • namespace URI will be empty • localName will be the name as given in the XML document • qName will be the name as given in the XML document If there is a default Namespace in effect and an element name has no prefix: • namespaceURI will be the URI of the default Namespace • localName will be the name as given in the XML doc • qName will be the name as given in the XML doc (there is no prefix, since the default Namespace doesn't specify a prefix)	Whether or not an element's name has a namespace prefix: • namespaceURI will be empty • localName will be empty • qName will be the name as given in the XML document (including the prefix, if there is one)

Table 12–6 describes the effect of Namespace processing on Attributes data.

Table 12–6: Effect Of Namespace Processing On Attributes Data

If Namespace Processing Is Enabled...	If Namespace Processing Is Suppressed...
The behavior is the same as that for Element names, except that a default Namespace does not apply to attribute names, so an attribute with no prefix will never have a value for its namespaceURI.	Whether or not an attribute's name has a namespace prefix: • namespaceURI will be empty • localName will be empty • qName will be the name as given in the XML document (including any prefix)

For more information on Namespace processing, see the reference entry for the SUPPRESS-NAMESPACE-PROCESSING attribute in the *Progress Version 9 Product Update Bulletin*.

12.3.6 Parsing With One Call Or With Multiple Calls

A Progress SAX application can parse an XML document using one of the following techniques:

- Single call
- Progressive scan

To use the *single-call* technique, the application calls the SAX-PARSE() method once. The parser parses the entire XML document (unless errors occur), calling all appropriate callbacks, and returns control to the line in the code following SAX-PARSE().

To use the *progressive-scan* technique, the application calls the SAX-PARSE-FIRST() method once to initiate parsing, then calls the SAX-PARSE-NEXT() method repeatedly to parse each XML token in the document. As each XML token is detected, Progress invokes the corresponding callback. After each call to SAX-PARSE-FIRST() or SAX-PARSE-NEXT(), control returns to the line in the code following the SAX-PARSE-FIRST() or SAX-PARSE-NEXT().

Consider using progressive scan:

- If your business logic is complex and processes individual XML elements extensively

  To do significant processing with single call, your callback code might have to call directly into your business logic. This might be awkward — especially adding SAX to existing code.

  To do significant processing with progressive scan, your business logic can simply call SAX-PARSE-FIRST() or SAX-PARSE-NEXT() to extract the next piece of data. The callbacks would simply store incoming data, one piece at a time, for the business logic to process after the return from SAX-PARSE-FIRST() or SAX-PARSE-NEXT().

- To parse two XML sources concurrently

After calling SAX-PARSE(), SAX-PARSE-FIRST(), or SAX-PARSE-NEXT(), the application checks the value of the PARSE-STATUS attribute, as explained in the "Monitoring the State Of the Parse" section.

12.3.7 Monitoring the State Of the Parse

A Progress SAX application keeps track of the status of the parse by monitoring the value of the SAX-READER's PARSE-STATUS attribute, which can assume the following values:

- SAX-UNINITIALIZED
- SAX-RUNNING
- SAX-COMPLETE
- SAX-PARSER-ERROR

For more information, see the PARSE-STATUS Attribute reference entry in the *Progress Version 9 Product Update Bulletin*.

12.3.8 Error Handling

When a Progress SAX application parses an XML document, it might encounter one of three distinct parse-related error situations. They are:

1. SAX-PARSE(), SAX-PARSE-FIRST(), or SAX-PARSE-NEXT() detects an error and cannot complete its current parse operation.

   This might be caused by one of the following:

   - The specified XML source does not exist.
   - The handle to the procedure containing the callbacks is invalid.
   - SAX-READER was not in the appropriate state. This can happen, for example, if SAX-PARSE-NEXT() was called before SAX-PARSE-FIRST().

   If this error situation occurs:

   a. Parsing stops.

   b. Progress raises an error.

      If NO-ERROR was specified, Progress sets ERROR-STATUS:ERROR to YES, sets ERROR-STATUS:NUM-MESSAGES to the number of errors encountered, and returns a Progress error message in response to a statement with the following syntax:

      SYNTAX

      ```
      ERROR-STATUS:GET-MESSAGE(err-msg-num).
      ```

      where err-*msg-num* indicates a number between 1 and ERROR-STATUS:NUM-MESSAGES inclusive.

      If NO-ERROR was not specified, Progress looks in the driver routine for the closest block that has the error property and behaves as if the block has an explicit ON ERROR phrase.

2. A callback raises an error by using the RETURN statement's ERROR option.

   If this error situation occurs:

   a. Parsing stops.

   b. Progress raises an error.

      If NO-ERROR was specified, Progress sets ERROR-STATUS:ERROR to YES and sets ERROR-STATUS:NUM-MESSAGES to zero.

      If NO-ERROR was not specified, Progress looks in the driver routine for the closest block that has the error property and behaves as if the block has an explicit ON ERROR phrase.

   c. RETURN-VALUE is set to whatever string the RETURN statement included.

   NOTE: If a callback procedure calls a second procedure, the second procedure calls a third procedure, etc., and the final procedure in the chain executes RETURN ERROR, each preceding procedure in the chain must also execute RETURN ERROR, else the error condition is never communicated to the driver routine. This is standard Progress behavior.

3. While a callback is executing, Progress raises an error that the callback does not handle—for example, a FIND CUSTOMER that fails to find the specified customer.

   If this error situation occurs:

   a. The error is displayed (which is standard Progress behavior).

   b. In the callback, Progress finds the closest block that has the error property (which might be the PROCEDURE block) and behaves according to the block's explicit or implicit ON ERROR phrase. This might cause Progress to break out of the callback.

   c. The PARSE-STATUS attribute and the ERROR-STATUS handle are unaffected.

   d. The parse continues.

12.4 Developing Progress SAX Applications

The following sections tell you how to develop a Progress SAX application:

12.4.1 Basic Tasks Of a Progress SAX Application

A typical Progress SAX application does the following:

1. Creates a SAX-READER object
2. Runs a persistent procedure that contains the callbacks
3. Configures the SAX-READER object by:
    a. Setting its HANDLER attribute to the handle of the routine that contains the callbacks
    b. Turning namespace processing and validation on or off as desired
    c. Specifying the input source using the SET-INPUT-SOURCE() method
4. Starts the parser by calling SAX-PARSE() or SAX-PARSE-FIRST()
5. Handles XML data passed to the callbacks as the parser proceeds
6. Monitors the state of the parse by checking error codes and the parse status after each call to SAX-PARSE(), SAX-PARSE-FIRST(), and SAX-PARSE-NEXT()
7. Releases resources, including deleting the SAX-READER object

Most of these tasks can be performed using the attributes and methods of the SAX-READER object, which are summarized in Table 12–7.

Table 12–7: Tasks Handled By the Attributes and Methods of SAX-READER

To Perform This Task...	Use This Attribute Or Method...
Specify the XML input	SET-INPUT-SOURCE() Method
Specify a search path for the DTD	SCHEMA-PATH Attribute
Tell the parser where the callbacks reside	HANDLER Attribute
Toggle parser options	SUPPRESS-NAMESPACE-PROCESSING Attribute VALIDATION-ENABLED Attribute
Start, continue, or stop parsing	SAX-PARSE() Method SAX-PARSE-FIRST() Method SAX-PARSE-NEXT() Method STOP-PARSING() Method
Get the status of the parse	LOCATOR-COLUMN-NUMBER Attribute LOCATOR-LINE-NUMBER Attribute LOCATOR-PUBLIC-ID Attribute LOCATOR-SYSTEM-ID Attribute PARSE-STATUS Attribute
Get or set information concerning this particular SAX-READER object	PRIVATE-DATA Attribute

12.4.2 Example 1 — Retrieving Names and Phone Numbers

This example retrieves names and phone numbers. It is presented in two versions in the following sections:

- Without Namespace Processing
- With Namespace Processing

Without Namespace Processing

Here is the EPI SAX Example 1 driver, e-sax1d.p. The driver's logic closely parallels the tasks in the "Basic Tasks Of a Progress SAX Application" section.

e-sax1d.p *(1 of 2)*

```
DEF VAR hParser as HANDLE.
DEF VAR hHandler AS HANDLE.
```

```
/* create the SAX-READER object */
CREATE SAX-READER hParser.

/* run the persistent procedure that contains the callbacks */
RUN "e-sax1h.p" PERSISTENT SET hHandler.

/* give the SAX-READER the handle to the persistent procedure */
hParser:HANDLER = hHandler.

/* Give the SAX-READER the info on the file to parse.
   This XML file does not use namespaces. */
hParser:SET-INPUT-SOURCE("FILE", "e-saxe1.xml").
```

```
hParser:SAX-PARSE( ) NO-ERROR.

/* By the time SAX-PARSE returns, our callbacks have been called as many
times as necessary and we're done processing the XML document (or there
was an error) */
```

```
IF ERROR-STATUS:ERROR THEN
DO:
  IF ERROR-STATUS:NUM-MESSAGES > 0 THEN
    /* unable to begin the parse */
    MESSAGE ERROR-STATUS:GET-MESSAGE(1) VIEW-AS ALERT-BOX.
  ELSE
    /* error detected in a callback */
    MESSAGE RETURN-VALUE VIEW-AS ALERT-BOX.
END.
ELSE
  MESSAGE "Document parsed successfully" VIEW-AS ALERT-BOX.
```

e-sax1d.p *(2 of 2)*

```
DELETE OBJECT hParser.
DELETE PROCEDURE hHandler.
```

Here is the EPI SAX Example 1 XML file, e-sax1.xml. Each entry contains a name and phone number.

e-sax1.xml

```
<?xml version='1.0' ?>
<Phonelist>
  <Entry ContactName="Jane Jones"> 555 555-5555 </Entry>
  <Entry ContactName="John Smith"> 555 555-1111 </Entry>
</Phonelist>
```

Here is the EPI SAX Example 1 handler procedure, e-sax1h.p. It contains the callbacks.

e-sax1h.p *(1 of 2)*

```
/* This small example uses a very simple approach to keeping track of where
it is in the processing of the document. It uses currentPerson and
currentNum, which are variables global to this procedure that enable the
application to tie together the several different pieces of information
that it gets for each element in the XML document. */

/* Name attribute for the current entry. App gets it during the StartElement
callback */
DEF VAR currentPerson AS CHARACTER.

/* Phone number from the current entry. App gets it during the Characters
callback because it is the character data for the element. */
DEF VAR currentNum AS CHARACTER.
```

```
/* This procedure is called when the parser finds the start tag for an element.
For this particular XML doc, the app simply looks for “Entry” elements and
digs out the “ContactName” attribute during the StartElement call, saving
it in currentPerson */
PROCEDURE StartElement:
  DEFINE INPUT PARAMETER namespaceURI AS CHARACTER.
  DEFINE INPUT PARAMETER localName AS CHARACTER.
  DEFINE INPUT PARAMETER qname AS CHARACTER.
  DEFINE INPUT PARAMETER attributes AS HANDLE.

  IF qName = "Entry" THEN
    currentPerson = attributes:GET-VALUE-BY-QNAME("ContactName").

END.
```

e-sax1h.p *(2 of 2)*

```
/* This callback gets passed the character data for an element. Note that SAX
does not guarantee that all the characters for an element get passed in
one call -- that's why the app has to maintain the currentNum global
variable and append to it when handling Characters, and also why it has to
wait for EndElement before displaying the message box. (Note also that some
apps may need to use a MEMPTR to accumulate the character data, which may
exceed the 32K Progress CHARACTER variable limit) */
PROCEDURE Characters:
  DEFINE INPUT PARAMETER charData AS MEMPTR.
  DEFINE INPUT PARAMETER numChars AS INTEGER.
  /* ( Can assume that any call to Characters is for an Entry's text value,
    because we know what the document looks like. If this weren't the case,
    we'd have to keep track of the localName passed to the most recent call to
    StartElement) */
  currentNum = currentNum + GET-STRING(charData, 1, GET-SIZE(charData)).
END.
```

```
/* This callback is called when the parser finds the end tag for an Element.
Note that this app only cares about the end of an Entry element.*/
PROCEDURE EndElement:
  DEFINE INPUT PARAMETER namespaceURI AS CHARACTER.
  DEFINE INPUT PARAMETER localName AS CHARACTER.
  DEFINE INPUT PARAMETER qName AS CHARACTER.

  IF qName = "Entry" THEN
  DO:
    MESSAGE "Name: " currentPerson SKIP
            "Phone Number: " currentNum VIEW-AS ALERT-BOX.
    currentNum = "".
    currentPerson = "".
  END.
END.
```

```
/* note that, knowing the structure of the XML doc, the app could have done
this in the EndElement call for the Phonelist element and could then have
omitted EndDocument altogether. */
PROCEDURE EndDocument:
  MESSAGE "All Done" VIEW-AS ALERT-BOX.
END.
```

When Example 1 is run, it produces the trace shown in Table 12–8.

Table 12–8: Trace of Example 1 *(1 of 2)*

Callback function: StartDocument

Callback function: StartElement
  namespaceURI:
  localName: Phonelist
  qName: Phonelist
  SAX-ATTRIBUTE has 0 items:

Callback function: StartElement
  namespaceURI:
  localName: Entry
  qName: Entry
  SAX-ATTRIBUTE has 1 items:
  Attribute 1 :
  namespaceURI:
  localName: ContactName
  qName: ContactName
  type: CDATA
  value: Jane Jones

Callback function: Characters   charData: 555 555-5555

Callback function: EndElement
  namespaceURI:
  localName: Entry
  qName: Entry

Callback function: StartElement
  namespaceURI:
  localName: Entry
  qName: Entry
  SAX-ATTRIBUTE has 1 items:
  Attribute 1 :
  namespaceURI:
  localName: ContactName
  qName: ContactName
  type: CDATA
  value: John Smith

Callback function: Characters   charData: 555 555-1111

Table 12–8: Trace of Example 1 *(2 of 2)*

```
Callback function: EndElement
  namespaceURI:
  localName: Entry
  qName: Entry
```

```
Callback function: EndElement
  namespaceURI:
  localName: Phonelist
  qName: Phonelist
```

```
Callback function: EndDocument
```

With Namespace Processing

Here is the same example except that the XML document uses Namespaces. Consequently, the StartElement and EndElement callbacks in the handler use the namespaceURI and localName parameters rather than qName.

NOTE: The original example could have used localName by itself, but did not.

Here is the EPI SAX Example 1 driver with namespace processing, e-sax1dn.p.

e-sax1dn.p *(1 of 2)*

```
DEF VAR hParser as HANDLE.
DEF VAR hHandler AS HANDLE.
```

```
/* create the SAX-READER object */
CREATE SAX-READER hParser.

/* run the persistent procedure that contains the callbacks */
RUN "e-saxe1h-ns.p" PERSISTENT SET hHandler.

/* give the SAX-READER the handle to the persistent procedure */
hParser:HANDLER = hHandler.

/* Give the SAX-READER the info on the file to parse.
   This XML file uses namespaces. */
hParser:SET-INPUT-SOURCE("FILE", "e-saxe1-ns.xml").
```

```
hParser:SAX-PARSE( ) NO-ERROR.
```

e-sax1dn.p *(2 of 2)*

```
/* By the time SAX-PARSE returns, our callbacks have been called as many
times as necessary and we're done processing the XML document (or there
was an error) */

IF ERROR-STATUS:ERROR THEN
DO:
  IF ERROR-STATUS:NUM-MESSAGES > 0 THEN
    /* unable to begin the parse */
    MESSAGE ERROR-STATUS:GET-MESSAGE(1) VIEW-AS ALERT-BOX.
  ELSE
    /* error detected in a callback */
    MESSAGE RETURN-VALUE VIEW-AS ALERT-BOX.
END.
ELSE
  MESSAGE "Document parsed successfully" VIEW-AS ALERT-BOX.
```

```
DELETE OBJECT hParser.
DELETE PROCEDURE hHandler.
```

Here is the EPI SAX Example 1 XML file with namespace processing, e-sax1n.xml.

e-sax1n.xml

```
<?xml version='1.0' ?>
<Phonelist xmlns="http:/www.wmhwmh.biz/ns/Default"
           xmlns:pl="http://www.wmhwmh.biz/ns/phonelist">
  <pl:Entry pl:ContactName="Jane Jones"> 555 555-5555 </pl:Entry>
  <pl:Entry pl:ContactName="John Smith"> 555 555-1111 </pl:Entry>
</Phonelist>
```

Here is the EPI SAX Example 1 handler procedure with namespace processing, e-sax1hn.p.

e-sax1hn.p *(1 of 3)*

```
/* Name attribute for the current entry. App gets it during the StartElement
callback */
DEF VAR currentPerson AS CHARACTER.

/* Phone number from the current entry. App gets it during the Characters
callback because it is the character data for the element. */
DEF VAR currentNum AS CHARACTER.
```

e-sax1hn.p *(2 of 3)*

```
/* This procedure is called when the parser finds the start tag for an element.
  For this particular XML doc, the app simply looks for "Entry" elements and
  digs out the "ContactName" attribute during the StartElement call, saving it
  in currentPerson.
  The code assumes that Namespace processing is enabled and checks
  to make sure that name parameters are part of the correct namespace
  */
PROCEDURE StartElement:
  DEFINE INPUT PARAMETER namespaceURI AS CHARACTER.
  DEFINE INPUT PARAMETER localName AS CHARACTER.
  DEFINE INPUT PARAMETER qName AS CHARACTER.
  DEFINE INPUT PARAMETER attributes AS HANDLE.

  IF namespaceURI = "http://www.wmhwmh.biz/ns/phonelist" THEN
  DO:
     IF localName = "Entry" THEN
        currentPerson = attributes:GET-VALUE-BY-NAMESPACE-NAME(
                                        "http://www.wmhwmh.biz/ns/phonelist",
                                        "ContactName" ).
  END.
END.
```

```
/* This callback gets passed the character data for an element. SAX
does not guarantee that all the characters for an element get passed in
one call -- that's why the app has to maintain the currentNum global
variable and append to it when handling Characters, and also why it has to
wait for EndElement before displaying the message box. (Some apps may
need to use a MEMPTR to accumulate the character data, which may
exceed the 32K Progress CHARACTER variable limit) */
PROCEDURE Characters:
  DEFINE INPUT PARAMETER charData AS MEMPTR.
  DEFINE INPUT PARAMETER numChars AS INTEGER.
  /* ( Can assume that any call to Characters is for an Entry's text value,
    because we know what the document looks like. If this weren't the case,
    we'd have to keep track of the localName passed to the most recent call to
    StartElement) */
  currentNum = currentNum + GET-STRING(charData, 1, GET-SIZE(charData)).
END.
```

e-sax1hn.p *(3 of 3)*

```
/* This callback is called when the parser finds the end tag for an Element.
This app only cares about the end of an Entry element.*/
PROCEDURE EndElement:
  DEFINE INPUT PARAMETER namespaceURI AS CHARACTER.
  DEFINE INPUT PARAMETER localName AS CHARACTER.
  DEFINE INPUT PARAMETER qName AS CHARACTER.

  IF namespaceURI = "http://www.wmhwmh.biz/ns/phonelist" THEN
  DO:
     IF  localName = "Entry" THEN
     DO:
       MESSAGE "Name: " currentPerson SKIP
            "Phone Number: " currentNum VIEW-AS ALERT-BOX.
       currentNum = "".
       currentPerson = "".
     END.
  END.
END.
```

```
/* note that, knowing the structure of the XML doc, the app could have done
this in the EndElement call for the Phonelist element and could then have
omitted EndDocument altogether. */
PROCEDURE EndDocument:
  MESSAGE "All Done" VIEW-AS ALERT-BOX.
END.
```

When Example 1 with namespace processing is run, it produces the trace shown in Table 12–9.

Table 12–9: Trace of Example 1 With Namespace Processing *(1 of 2)*

```
Callback function: StartDocument
```

```
Callback function: StartElement
  namespaceURI: http:/www.wmhwmh.biz/ns/Default
  localName: Phonelist
  qName: Phonelist
  SAX-ATTRIBUTE has 0 items:
```

```
Callback function: StartElement
  namespaceURI: http://www.wmhwmh.biz/ns/phonelist
  localName: Entry
  qName: pl:Entry
  SAX-ATTRIBUTE has 1 items:
    Attribute 1 :
      namespaceURI: http://www.wmhwmh.biz/ns/phonelist
      localName: ContactName
      qName: pl:ContactName
      type: CDATA
      value: Jane Jones
```

```
Callback function: Characters
  charData:  555 555-5555
```

```
Callback function: EndElement
  namespaceURI: http://www.wmhwmh.biz/ns/phonelist
  localName: Entry
  qName: pl:Entry
```

```
Callback function: StartElement
  namespaceURI: http://www.wmhwmh.biz/ns/phonelist
  localName: Entry
  qName: pl:Entry
  SAX-ATTRIBUTE has 1 items:
    Attribute 1 :
      namespaceURI: http://www.wmhwmh.biz/ns/phonelist
      localName: ContactName
      qName: pl:ContactName
      type: CDATA
      value: John Smith
```

```
Callback function: Characters
  charData:  555 555-1111
```

Table 12–9: Trace of Example 1 With Namespace Processing *(2 of 2)*

```
Callback function: EndElement
  namespaceURI: http://www.wmhwmh.biz/ns/phonelist
  localName: Entry
  qName: pl:Entry
```

```
Callback function: EndElement
  namespaceURI: http:/www.wmhwmh.biz/ns/Default
  localName: Phonelist
  qName: Phonelist
```

```
Callback function: EndDocument
```

12.4.3 Example 2 — Reading Customer Data and Writing a TEMP-TABLE

This example is a SAX version of the DOM example described in Chapter 11, "XML Support," in this book. The example reads an XML file containing the Customer table from the Sports database and writes the data to a temp-table. The example uses qname, assumes there is no namespace prefix, and, for clarity, omits code for transaction scoping and for validation.

Here is the EPI SAX Example 2 driver procedure, e-sax2d.p.

e-sax2d.p

```
DEF VAR hParser as HANDLE.
DEF VAR hHandler AS HANDLE.

/* create the SAX-READER object */
CREATE SAX-READER hParser.

/* run the persistent procedure that contains the callbacks */
RUN "e-sax2h.p" PERSISTENT SET hHandler.

/* give the SAX-READER the handle to the persistent procedure */
hParser:HANDLER = hHandler.

/* Give the SAX-READER the info on the file to parse.
   This XML file does not use namespaces. */
hParser:SET-INPUT-SOURCE("FILE", "e-saxe2.xml").

hParser:SAX-PARSE( ) NO-ERROR.

/* By the time SAX-PARSE returns, our callbacks have been called as many
times as necessary and we're done processing the XML document (or there
was an error) */

IF ERROR-STATUS:ERROR THEN
DO:
  IF ERROR-STATUS:NUM-MESSAGES > 0 THEN
    /* unable to begin the parse */
    MESSAGE ERROR-STATUS:GET-MESSAGE(1) VIEW-AS ALERT-BOX.
  ELSE
    /* error detected in a callback */
    MESSAGE RETURN-VALUE VIEW-AS ALERT-BOX.
END.
ELSE
  MESSAGE "Document parsed successfully" VIEW-AS ALERT-BOX.

DELETE OBJECT hParser.
DELETE PROCEDURE hHandler.
```

Here is the EPI SAX Example 2 XML file, e-sax2.xml.

e-sax2.xml

```
<?xml version='1.0' ?>
<Customers>
```

```
 <Customer Name="Lift Line Skiing" Cust-num="1">
    <Country>USA</Country>
    <Address>276 North Street</Address>
    <Address2></Address2>
    <City>Boston</City>
    <State>MA</State>
    <Postal-Code>02114</Postal-Code>
    <Contact>Gloria Shepley</Contact>
    <Phone>(617) 450-0087</Phone>
    <Sales-Rep>HXM</Sales-Rep>
    <Credit-Limit>66700</Credit-Limit>
    <Balance>42568</Balance>
    <Terms>Net30</Terms>
    <Discount>35</Discount>
    <Comments>This customer is on credit hold.</Comments>
  </Customer>
```

```
 <Customer Name="Hoops " Cust-num="3">
     <Country>USA</Country>
     <Address>Suite 415</Address>
     <Address2>40 Grove St.</Address2>
     <City>Atlanta</City>
     <State>GA</State>
     <Postal-Code>02112</Postal-Code>
     <Contact>Michael Traitser</Contact>
     <Phone>(617) 355-1557</Phone>
     <Sales-Rep>HXM</Sales-Rep>
     <Credit-Limit>75000</Credit-Limit>
     <Balance>1199.95</Balance>
     <Terms>Net30</Terms>
     <Discount>10</Discount>
     <Comments>This customer is now OFF credit hold.</Comments>
</Customer>
```

```
</Customers>
```

NOTE: There is no DTD and no use of namespace prefixes. The lack of a DTD means that the handlers need to validate the document, but the example omits that validation for the sake of clarity.

Here is the EPI SAX Example 2 handler procedure, e-sax2h.p.

e-sax2h.p *(1 of 3)*

```
/*so we can create new records*/
DEFINE TEMP-TABLE Custt LIKE Customer.
DEFINE VARIABLE hBuf AS HANDLE.
DEFINE VARIABLE hDBFld AS HANDLE.
hBuf = BUFFER Custt:HANDLE.

/* variable in which to accumulate all the text data for one element
coming in through potentially multiple calls (per element) to the
Characters procedure */
DEF VAR currentFieldValue as CHAR.

/* simple-minded state machine - the code makes minimal use of it,
but it could easily be used to validate the structure of the document
in this example */
DEF VAR iProcessingState as INT.
&SCOPED-DEFINE READY-TO-START 1
&SCOPED-DEFINE GETTING-RECORDS 2
&SCOPED-DEFINE GETTING-FIELDS 3
&SCOPED-DEFINE DONE 4
```

```
PROCEDURE StartDocument:
  iProcessingState = {&READY-TO-START}.
END.
```

e-sax2h.p *(2 of 3)*

```
PROCEDURE StartElement:
  DEFINE INPUT PARAMETER namespaceURI AS CHARACTER.
  DEFINE INPUT PARAMETER localName AS CHARACTER.
  DEFINE INPUT PARAMETER qName AS CHARACTER.
  DEFINE INPUT PARAMETER attributes AS HANDLE.

  IF qName = "Customers" THEN
    iProcessingState = {&GETTING-RECORDS}.
  ELSE IF qName = "Customer" THEN
  DO:
    /* starting a new customer, so create the record */
    CREATE Custt.

    /*get the fields that are in the XML doc as attributes*/
    cust-num = INTEGER( attributes:GET-VALUE-BY-QNAME("Cust-num") ).

    name = attributes:GET-VALUE-BY-QNAME( "Name" ).
    iProcessingState = {&GETTING-FIELDS}.
  END.
  ELSE IF iProcessingState = {&GETTING-FIELDS} THEN
  DO:
    /* get a handle to the field whose name corresponds
       to the element name */
    hDBFld = hBuf:BUFFER-FIELD(qName).

    /* re-init the variable in which we accumulate the field value */
    currentFieldValue = "".
  END.
END.
```

```
PROCEDURE Characters:
  DEFINE INPUT PARAMETER charData AS MEMPTR.
  DEFINE INPUT PARAMETER numChars AS INTEGER.

  /* get the text value of the field (hDBFld was set to the correct field
     in StartElement */
  currentFieldValue =
      currentFieldValue + GET-STRING(charData, 1, GET-SIZE(charData)).
END.
```

e-sax2h.p *(3 of 3)*

```
PROCEDURE EndElement:
  DEFINE INPUT PARAMETER namespaceURI AS CHARACTER.
  DEFINE INPUT PARAMETER localName AS CHARACTER.
  DEFINE INPUT PARAMETER qName as CHARACTER.

  IF localName = "Customers" THEN
    iProcessingState = {&DONE}.
  ELSE IF localName = "Customer"THEN
    iProcessingState = {&GETTING-RECORDS}.
  ELSE IF iProcessingState = {&GETTING-FIELDS} THEN
    hDBFld:BUFFER-VALUE = currentFieldValue.

END.
```

```
PROCEDURE EndDocument:
  /* show that data made it by displaying temp-table */
  FOR EACH Custt:
    DISPLAY custt.
  END.
  RUN Cleanup.
END.
```

```
PROCEDURE FatalError:
  DEFINE INPUT PARAMETER errMessage AS CHARACTER.

  /* not necessary to do anything with PRIVATE-DATA, this is
     just an example of what you could do */
  SELF:PRIVATE-DATA = "FATAL".
  RUN Cleanup.

  /* RETURN ERROR in an error handler implicitly calls SELF:STOP-PARSING(),
     sets SELF:PARSE-STATUS to SAX-PARSER-ERROR, and raises the Progress
     ERROR condition */
  RETURN ERROR errMessage
            + "(Line " + CHR(SELF:LOCATOR-LINE-NUMBER)
            + ", Col " + CHR(SELF:LOCATOR-COLUMN-NUMBER)
            + ")".
END.
```

```
/* this is not a SAX callback, it is just a local utility */
PROCEDURE Cleanup:
  /* in case we've parsed previous documents */
  hBuf:EMPTY-TEMP-TABLE( ).
END.
```

12.4.4 Progress SAX and WebSpeed

Using SAX with WebSpeed applications is relatively straightforward. Just be sure to perform the following extra steps:

1 ♦ Check that WEB-CONTEXT's IS-XML attribute is YES. This indicates that the WebSpeed transaction server recognizes that an XML document was posted to it.

NOTE: WEB-CONTEXT's VALIDATE-XML attribute applies only to DOM, not to SAX.

2 ♦ After you create the SAX-READER object, run the SET-INPUT-SOURCE() method as follows:

```
hSAX-READER:SET-INPUT-SOURCE("HANDLE", WEB-CONTEXT).
```

At this point, proceed with the WebSpeed application as if it were any other Progress SAX application.

12.4.5 Example 3 — Reading XML Data Using WebSpeed

This example reads XML data using WebSpeed.

The example can use the callbacks in e-sax2h.p, the EPI SAX Example 2 handler procedure.

The example consists of e-saxe3s.p, the EPI SAX Example 3 server procedure.

e-saxe3s.p *(1 of 2)*

```
/* This particular .p is intended to be run on a server with an
   available webserver and functioning WebSpeed broker/messenger */
DEF VAR hParser AS HANDLE.
DEF VAR hHandler AS HANDLE.
CREATE SAX-READER hParser.
```

```
/************************************************************
*
* This is needed to support webspeed applications
*
************************************************************/
{src/web/method/cgidefs.i}
```

```
/* run the persistent procedure that contains the callbacks */
RUN "e-saxe2h.p" PERSISTENT SET hHandler.
```

```
/* give the SAX-READER the handle to the persistent procedure */
hParser:HANDLER =  hHandler.
```

```
/************************************************************
*
* Goal: check to see if there is a xml document available on the
* webstream and if yes give it to the sax parser.
*
************************************************************/
IF (WEB-CONTEXT:IS-XML) THEN
   hParser:SET-INPUT-SOURCE("handle", WEB-CONTEXT).
```

```
hParser:SAX-PARSE( ) NO-ERROR.
/* By the time SAX-PARSE returns, our callbacks have been called as many
   times as necessary and we're done processing the XML document
   (or there was an error) */
```

e-saxe3s.p *(2 of 2)*

```
IF ERROR-STATUS:ERROR THEN
DO:
   IF ERROR-STATUS:NUM-MESSAGES > 0 THEN
      /* unable to begin the parse */
      MESSAGE ERROR-STATUS:GET-MESSAGE(1) VIEW-AS ALERT-BOX.
   ELSE
      /* error raised in a callback */
      MESSAGE RETURN-VALUE VIEW-AS ALERT-BOX.
   END.
ELSE
    MESSAGE "Document parsed successfully".
```

```
DELETE OBJECT hParser.
DELETE PROCEDURE hHandler.
```

12.4.6 SAX and the Progress ADE

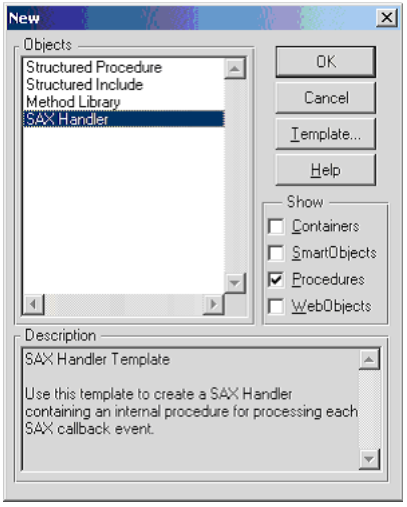
You can use the Progress ADE to develop Progress SAX applications. This involves the following tasks:

- Creating a SAX Handler object
- Supplying an override for each callback your application requires
- Handling context (optional)

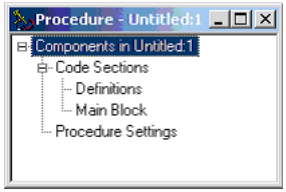
To create a SAX Handler object, which corresponds to a procedure (`.p`) file to contain the SAX callbacks, follow these steps:

1 ♦ From the Progress AppBuilder main menu, select **File** → **New**. The New dialog box appears.

2 ♦ Select the **Procedures** toggle box. The SAX handler template appears, as shown here:



3 ♦ Select Sax Handler and click **OK**. The new Sax Handler object appears as follows:

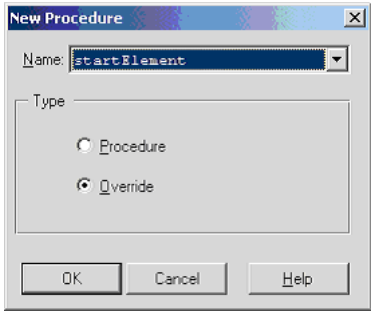


To supply the callbacks that your SAX application requires, follow these steps:

1 ♦ On the AppBuilder tool bar, click the **Edit Code** icon, as shown here:

The **Section** combo box appears.

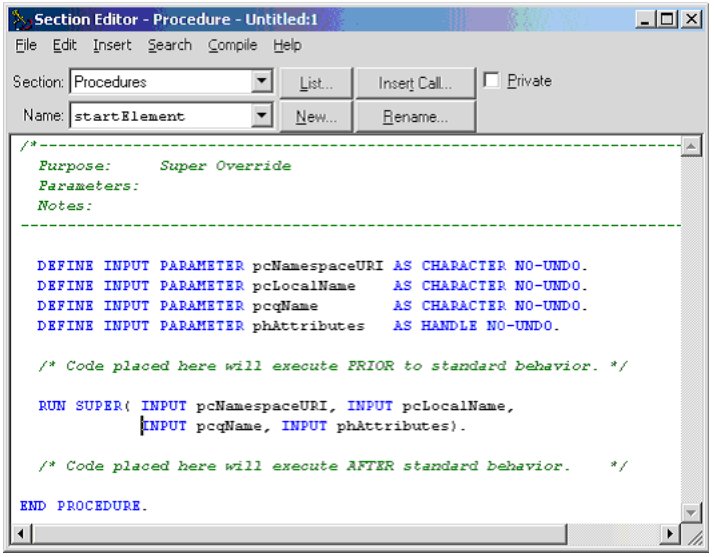
2 ♦ Change the **Section** combo box to **Procedure**. The New Procedure window appears, as shown here:



3 ♦ Select the name of the callback desired, select **Override**, then click **OK**.

NOTE: ADE implements SAX callbacks as super procedures (which you can override) of the SAX Handler object.

The Section Editor appears, as shown here:



4 ♦ Modify the callback as desired, then save it.

NOTE: If you misspell the name of a callback, at run time, it will not be invoked. Rather, the corresponding internal procedure in the super procedure will be invoked.

Storing and Recalling Context Information

The SAX specification does not say how to store and recall context information — that is, information on how XML elements are related to each other. For example, the SAX specification says that when the SAX parser encounters a new element, a startElement event should be triggered and the startElement callback should be invoked. However, the SAX specification does not say how to determine the new element's parent.

But Progress SAX provides a solution. Three of the Progress ADE templates for SAX callbacks refer to a temp-table. The temp-table and its records can be used as a stack to record context information related to that callback. When this feature is turned on:

- Each time the SAX parser encounters the beginning of a new element, the ADE creates a new temp-table record.
- Each time the SAX parser encounters the end of the element, the ADE deletes the temp-table record.

The information recorded in each temp-table record includes the parameters passed to startElement and the element's path (position) in the element hierarchy. For example, in the following XML example, the path of the customer element is `/customer` and the path of the order element is `/customer/orders/order`:

```
<customer custnum="1" name="Lift Line Skiing">
   <orders>
      <order ordernum="1">
      </order>
   </orders>
</customer>
```

To activate context management (which is inactive by default), call the setContextMode() function, as demonstrated in the following code fragment:

```
RUN myHandler.p PERSISTENT SET hHandler.
DYNAMIC-FUNCTION("setContextMode" IN hHandler, TRUE).
hParser:HANDLER = hHandler.
```

Progress SAX provides context management for the following SAX callbacks:

- StartDocument
- StartElement
- EndElement

Context Management Example

Here is a fragment that demonstrates the ADE context management system. The fragment retrieves the handle to the context management table, then finds the element added most recently:

```
PROCEDURE getTopElement :
  DEFINE OUTPUT PARAMETER cElementname AS CHARACTER.

  mhStack = DYNAMIC-FUNCTION("getStackHandle").
  IF VALID-HANDLE(mhStack) THEN DO:
    DEFINE VARIABLE fld AS HANDLE.
    DEFINE VARIABLE bh AS HANDLE.
    bh = mhStack:DEFAULT-BUFFER-HANDLE.
    bh:FIND-LAST() NO-ERROR.
    fld = bh:BUFFER-FIELD("cQName") NO-ERROR.
    cElementname = fld:BUFFER-VALUE NO-ERROR.
  END.

END PROCEDURE.
```

12.5 SAX API Reference

This section contains the following sections:

- SAX Error Message Reference
- SAX Callback Reference

For definitions of the 4GL elements related to the SAX-READER and SAX-ATTRIBUTES objects, see the *Progress Version 9 Product Update Bulletin*.

12.5.1 SAX Error Message Reference

Table 12–10 explains the error messages that Progress SAX provides.

Table 12–10: Progress SAX Error Messages

Error Message	Explanation
Couldn't initialize proxml (or libproxml).	`proxml.dll` (or `libproxml.so`) was missing or incomplete, or XML could not be initialized.
Parser not running for SAX-PARSE-NEXT.	Could not read next part of XML document: parser not running.
Document not found.	*`file-name`* was not found.
Handler procedure not found.	Could not process XML document: invalid procedure handle for the handler.

12.5.2 SAX Callback Reference

This section contains a reference entry for each callback Progress SAX supports.

Each entry specifies the signature and defines each parameter.

The callbacks, in alphabetical order, are:

- Characters
- EndDocument
- EndElement
- EndPrefixMapping
- Error
- FatalError
- IgnorableWhitespace
- NotationDecl
- ProcessingInstruction
- ResolveEntity
- StartDocument
- StartElement
- StartPrefixMapping
- UnparsedEntityDecl
- Warning

These callbacks closely match those defined in the SAX2 documentation at `www.saxproject.org`. For more information on these callbacks, see this Web site.

Characters

Invoked when the XML parser detects character data.

SYNTAX

```
PROCEDURE Characters:
  DEFINE INPUT PARAMETER charData AS MEMPTR.
  DEFINE INPUT PARAMETER numChars AS INTEGER.
```

charData

A MEMPTR that contains a chunk of character data.

numChars

The number of characters contained in the MEMPTR.

NOTE: If a character requires more than one byte to encode, the value of numChars might not match the value returned by MEMPTR:GETSIZE().

The parser calls this method to report each chunk of character data. It might report contiguous character data in one chunk, or split it into several chunks. If validation is enabled, whitespace is reported by the IgnorableWhitespace callback.

Although it is not intended for the application to call Characters directly, the application can. Whoever calls Characters must free the charData MEMPTR. When Progress calls Characters, Progress is responsible for freeing the charData MEMPTR (although if the application frees it, no harm results). If the application calls Characters, the application is responsible for freeing the charData MEMPTR.

To copy the charData MEMPTR such that the memory used by the copy is completely separate from the memory used by the original, use 4GL assignment, which performs a deep copy. The following fragment demonstrates this:

```
memptrA = memptrB
```

For more information on 4GL assignment, see the *Progress Language Referen*ce.

EndDocument

Invoked when the XML parser detects the end of an XML document.

SYNTAX

```
PROCEDURE EndDocument:
```

EndElement

Invoked when the XML parser detects the end of an element.

SYNTAX

```
PROCEDURE EndElement:
  DEFINE INPUT PARAMETER namespaceURI AS CHARACTER.
  DEFINE INPUT PARAMETER localName    AS CHARACTER.
  DEFINE INPUT PARAMETER qName        AS CHARACTER.
```

namespaceURI

A CHARACTER string indicating the namespace URI of the element.

If namespace processing is not enabled, or the element is not part of a namespace, the string is of length zero.

localName

A CHARACTER string indicating the nonprefixed element name.

If namespace processing is not enabled, the string is of length zero.

qName

A CHARACTER string indicating the actual name of the element in the XML source.

If the name has a prefix, *qName* includes it, whether or not namespace processing is enabled.

This method corresponds to a preceding StartElement after all element content is reported.

EndPrefixMapping

Invoked when the XML parser detects that a prefix associated with namespace mapping has gone out of scope.

SYNTAX

```
PROCEDURE EndPrefixMapping:
  DEFINE INPUT PARAMETER prefix AS CHARACTER.
```

prefix

A character string representing the prefix for a namespace declaration.

This callback is invoked only when namespace processing is enabled. It provides information not required by normal namespace processing. However, in some situations, this callback might be useful and even required.

Error

Invoked to report an error encountered by the parser while parsing the XML document.

SYNTAX

```
PROCEDURE Warning:
  DEFINE INPUT PARAMETER errMessage AS CHARACTER.
```

errMessage

A character string indicating the error message.

After this callback is invoked, the parser continues where it left off.

FatalError

Invoked to report a fatal error.

SYNTAX

```
PROCEDURE FatalError:
  DEFINE INPUT PARAMETER errMessage AS CHARACTER.
```

errMessage

A character string indicating the error message.

The application must assume that after a fatal error is reported, the document is unusable and future parsing events might not be reported. However, the parser might try to continue to parse the document. To stop the parser after reporting a fatal error, execute RETURN ERROR.

NOTE: If you stop the parser by executing STOP-PARSING(), parsing stops, but no error condition is raised, no error message is reported, SAX-READER's PARSE-STATUS attribute is set to SAX-COMPLETE rather than to SAX-PARSER-ERROR, and the driver might not know that an error occurred. For this reason, Progress Software Corporation recommends that to stop the parser after reporting a fatal error, execute RETURN ERROR.

IgnorableWhitespace

Invoked when the XML parser detects ignorable whitespace.

SYNTAX

```
PROCEDURE IgnorableWhitespace:
  DEFINE INPUT PARAMETER charData AS CHARACTER.
  DEFINE INPUT PARAMETER numChars AS INTEGER.
```

charData

A CHARACTER string representing a contiguous block of ignorable whitespace in an XML document.

numChars

An INTEGER expression indicating the size, in characters, of the character string.

If validation is enabled, the XML parser reports ignorable whitespace through this callback. If validation is not enabled, the XML parser reports whitespace through the Characters callback.

The data type of *charData* is CHARACTER, not MEMPTR, because it is unlikely that an XML document has over 32K of contiguous ignorable whitespace.

NotationDecl

Invoked when the XML parser detects a notation declaration.

SYNTAX

```
PROCEDURE NotationDecl:
  DEFINE INPUT PARAMETER name     AS CHARACTER.
  DEFINE INPUT PARAMETER publicID AS CHARACTER.
  DEFINE INPUT PARAMETER systemID AS CHARACTER.
```

name

A character string representing the name of the notation.

publicID

Optional. A character string indicating the public identifier of the entity.

If none is supplied, the string is of length zero.

systemID

Optional. A character string indicating the system identifier of the entity.

If none is supplied, the string is of length zero.

systemID must be one of the following:

- Absolute file path
- Relative file path
- Absolute URI

ProcessingInstruction

Invoked when the XML parser detects a processing instruction.

SYNTAX

```
PROCEDURE ProcessingInstruction:
  DEFINE INPUT PARAMETER target AS CHARACTER.
  DEFINE INPUT PARAMETER data   AS CHARACTER.
```

target

A character string indicating the target of the processing instruction.

data

A character string indicating the data associated with the processing instruction.

If the processing instruction has no data, the length of the string is zero.

NOTE: A processing instructions can appear before or after a root element.

ResolveEntity

Invoked to let the application specify the location of an external entity (such as a DTD).

When the parser finds an external entity reference, it calls ResolveEntity, passing it the system identifier and public identifier (if any) contained in the XML. This gives the application a chance to override the location specified in the XML.

NOTE: In ResolveEntity, you cannot use any I/O blocking statements, such as the UPDATE statement and the WAIT-FOR statement.

SYNTAX

```
PROCEDURE ResolveEntity:
  DEFINE INPUT  PARAMETER publicID   AS CHARACTER.
  DEFINE INPUT  PARAMETER systemID   AS CHARACTER.
  DEFINE OUTPUT PARAMETER filePath   AS CHARACTER.
  DEFINE OUTPUT PARAMETER memPointer AS MEMPTR.
```

publicID

Optional. A character string indicating the public identifier of the entity. If none is supplied, the string is of length zero.

systemID

A character string indicating the system identifier of the entity.

The character string will not be of length zero, as this parameter is required.

systemID will be one of the following:

- Absolute file path
- Relative file path
- Absolute URL

filePath

Optional. A character string indicating the actual location of the entity being resolved. This tells the parser where to actually get the entity, in preference to the location specified by the system identifier.

filePath will be one of the following:

- Absolute file path
- Relative file path
- HTTP URI

If you do not supply *filePath*, set it to ? (the unknown value).

memPointer

Optional. A MEMPTR containing the entity being resolved. Use *memPointer* to return XML representing an entity that is not stored as a stand-alone file.

If you do not supply *memPointer*, set it to ? (the unknown value).

CAUTION: Supplying both *filePath* and *memPointer* is an error.

If the application does not implement this callback, or if the callback sets both filePath and memPointer to ? (the unknown value), the entity is resolved according to the following rules (which are also the rules that the Progress XML DOM interface uses):

1. If the location given in the XML source is a relative path and the SAX-READER:SCHEMA-PATH attribute has been set, try appending the relative path to each entry in SCHEMA-PATH and retrieving the file there.

2. If the location is a relative file path and the SAX-READER:SCHEMA-PATH attribute has ? (the unknown value), try retrieving the file relative to the working directory.

3. If the location given in the XML source is an absolute path to a local file or if it is an HTTP URI, try retrieving the file at the specified location.

4. If the file cannot be found, Progress calls the FatalError callback (if there is one) and stops processing the XML.

StartDocument

Invoked when the XML parser detects the start of an XML document.

SYNTAX

```
PROCEDURE StartDocument:
```

StartDocument does not provide any data.

StartElement

Invoked when the XML parser detects the beginning of an element.

SYNTAX

```
PROCEDURE StartElement:
  DEFINE INPUT PARAMETER namespaceURI AS CHARACTER.
  DEFINE INPUT PARAMETER localName    AS CHARACTER.
  DEFINE INPUT PARAMETER qName        AS CHARACTER.
  DEFINE INPUT PARAMETER attributes   AS HANDLE.
```

namespaceURI

A character string indicating the namespace URI of the element.

If namespace processing is not enabled or the element is not part of a namespace, the string is of length zero.

localName

A character string indicating the non-prefixed element name.

If namespace processing is not enabled, the string is of length zero.

qName

A character string indicating the actual name of the element in the XML source.

If the name has a prefix, *qName* includes it, whether or not namespace processing is enabled.

attributes

A handle to a SAX-ATTRIBUTES object, which provides access to all attributes specified for the element.

If the element has no attributes, *attributes* is still a valid handle, and the NUM-ITEMS attribute is zero.

For every invocation of StartElement, there is a corresponding invocation of EndElement.

The contents of the element are reported in sequential order before the corresponding EndElement is invoked.

When StartElement returns, the SAX-ATTRIBUTES object, which was created by Progress, is deleted by Progress.

NOTE: If the application deletes it first, however, no harm is done.

StartPrefixMapping

Invoked when the XML parser detects that a prefix associated with namespace mapping is coming into scope.

NOTE: This callback is invoked only when namespace processing is enabled.

SYNTAX

```
PROCEDURE StartPrefixMapping:
  DEFINE INPUT PARAMETER prefix AS CHARACTER.
  DEFINE INPUT PARAMETER uri    AS CHARACTER.
```

prefix

A character string representing the prefix for a namespace declaration.

uri

A character string representing the URI that identifies the namespace being declared.

This callback does not normally need to be implemented, since the information it provides is not required for normal namespace processing. But it might be useful, and even required, in some situations.

UnparsedEntityDecl

Invoked when the XML parser detects an entity that it does not parse (where "unparsed entity" has the definition given in the XML 1.0 specification).

SYNTAX

```
PROCEDURE UnparsedEntityDecl:
  DEFINE INPUT PARAMETER name         AS CHARACTER.
  DEFINE INPUT PARAMETER publicID     AS CHARACTER.
  DEFINE INPUT PARAMETER systemID     AS CHARACTER.
  DEFINE INPUT PARAMETER notationName AS CHARACTER.
```

name

A character string indicating the name of the entity.

publicID

Optional. A character string indicating the public identifier of the entity.

If *publicID* is not supplied, the character string is of length zero.

systemID

Optional. A character string representing the system identifier of the entity.

If *systemID* is not supplied, the character string is of length zero.

systemID must be one of the following:

- Absolute file path
- Relative file path
- Absolute URI

notationName

A character string indicating the name of the notation associated with the entity.

Warning

Invoked to report a warning.

SYNTAX

```
PROCEDURE Warning:
  DEFINE INPUT PARAMETER errMessage AS CHARACTER.
```

errMessage

A character string indicating the error message.

A warning is a condition that is less severe than an error or a fatal error, as determined by the XML parser.

After this callback is invoked, the parser continues where it left off.

13

Accessing SonicMQ Messaging From the Progress 4GL

Application programmers can access Java Message Service (JMS) messaging from the Progress 4GL with the Progress SonicMQ Adapter. You can write 4GL applications that access the SonicMQ Adapter through a 4GL–JMS API that works the same on all platforms and in all configurations (GUI, character, AppServer, WebSpeed, and batch).

This chapter provides an introduction to accessing SonicMQ messaging service from the Progress 4GL and includes the following sections:

For more details about SonicMQ or JMS, see the *SonicMQ Programming Guide* and the *Java Message Service* specification, available on the Web at `http://www.java.sun.com`.

13.1 Introduction To the SonicMQ Adapter

The *Progress SonicMQ Adapter* is an adapter broker in the same administration framework as the Progress Explorer and the NameServer (see Figure 13–1). The SonicMQ Adapter works with a 4GL–JMS API to provide access to JMS messaging from 4GL applications. *Progress SonicMQ* is Progress Software's implementation of the Java Message Service (JMS) specification. *JMS* is set of interfaces and associated semantics that define, in a product independent manner, how Java clients can access Message-oriented Middleware (MOM) servers. *MOM* provides a reliable way for applications to create, send, receive, and read messages in a distributed enterprise system.

With the SonicMQ Adapter, 4GL programmers can use familiar Progress 4GL syntax and mechanisms to write JMS messaging applications, whether the application is GUI, character, AppServer, or WebSpeed. These applications can send messages to and receive messages from applications written in any other language. The integration point is Progress SonicMQ.

The 4GL–JMS API is strongly integrated with the 4GL programming model and style. Progress provides 4GL procedures that interact with a SonicMQ broker for JMS messaging in both domains, the two types of JMS messaging, Publish and Subscribe (Pub/Sub) and Point-to-Point (PTP). These procedures run persistently and represent the JMS connection, session, and Message objects. A 4GL programmer can use the methods in these objects for JMS message delivery, acknowledgement, and recovery.

The SonicMQ Adapter provides access to most SonicMQ features from the Progress 4GL. In a Progress 4GL-to-4GL messaging situation, an application can package 4GL data within standard messages, for example to send a temp table or a table.

With the 4GL–JMS API, all objects are persistent procedures, applications use the Progress 4GL event model, and 4GL applications can extend local publish and subscribe for distributed applications. Applications use the existing Progress 4GL error handling mechanisms to deal with 4GL–JMS errors. Messages are processed when the application is in a WAIT FOR or other IO blocking state. Non-UI applications, such as AppServer processes or batch processes that cannot use WAIT FOR, can use the waitForMessages 4GL–JMS API call, as can all GUI, character, AppServer, WebSpeed, and batch applications.

4GL applications written to take advantage of the 4GL–JMS API can talk with other applications without knowing whether the other application is a 4GL or non-4GL application. Java features are mapped to the Progress 4GL. For example, Java Enumeration Objects map to comma-separated lists in the Progress 4GL.

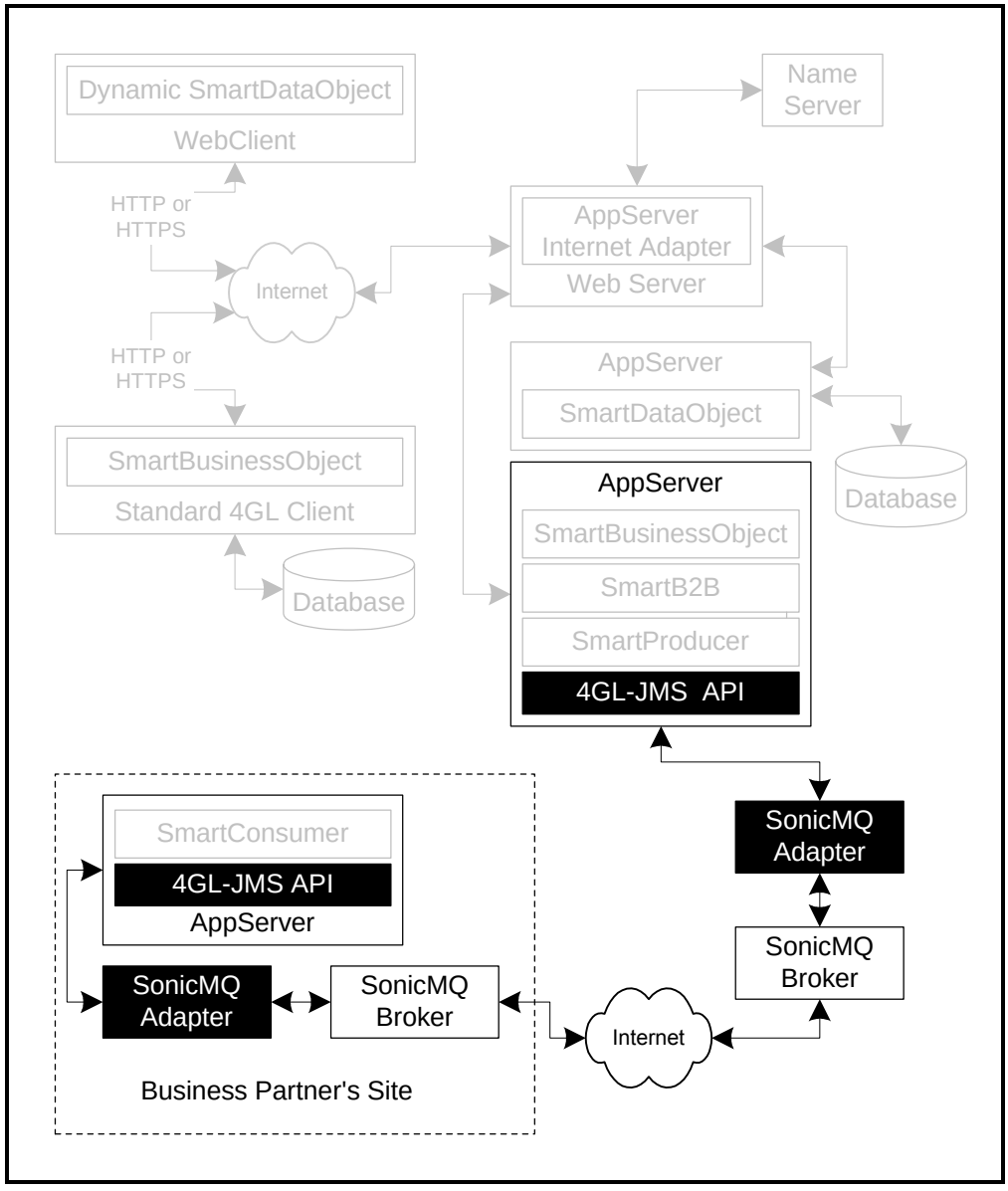


Figure 13–1: An Example Of the Progress SonicMQ Adapter In Context

13.1.1 SonicMQ Adapter Architecture

The SonicMQ Adapter belongs to the AppServer administration framework, which includes the Progress Explorer and NameServer. The SonicMQ Adapter can be started and configured by the Progress Explorer and from the command line. 4GL applications connect to the SonicMQ Adapter by specifying the connection parameters of the NameServer when the JMS session is created. Figure 13–1 shows one possible context for the SonicMQ Adapter. The application does not have to be on the Progress AppServer, nor is it even typical to have two SonicMQ Adapter brokers. The only software requirement on the 4GL side is a set of .r files that make the interface to the SonicMQ Adapter.

The installation process installs these files for the 4GL–JMS API: *<Progress_install_dir>*`/jms/*.r` and *<Progress_install_dir>*`/jms/impl/*.r`. As shown in Figure 13–2, each 4GL application connection has its own thread in the SonicMQ Adapter process. Only one broker process runs, but it is multi-threaded, with each 4GL client having its own thread running. These threads talk to SonicMQ. (This differentiates the SonicMQ Adapter from AppServer or WebSpeed, which both have one or more Progress process agents running.) It is also possible to start more than one SonicMQ Adapter process under a single NameServer.

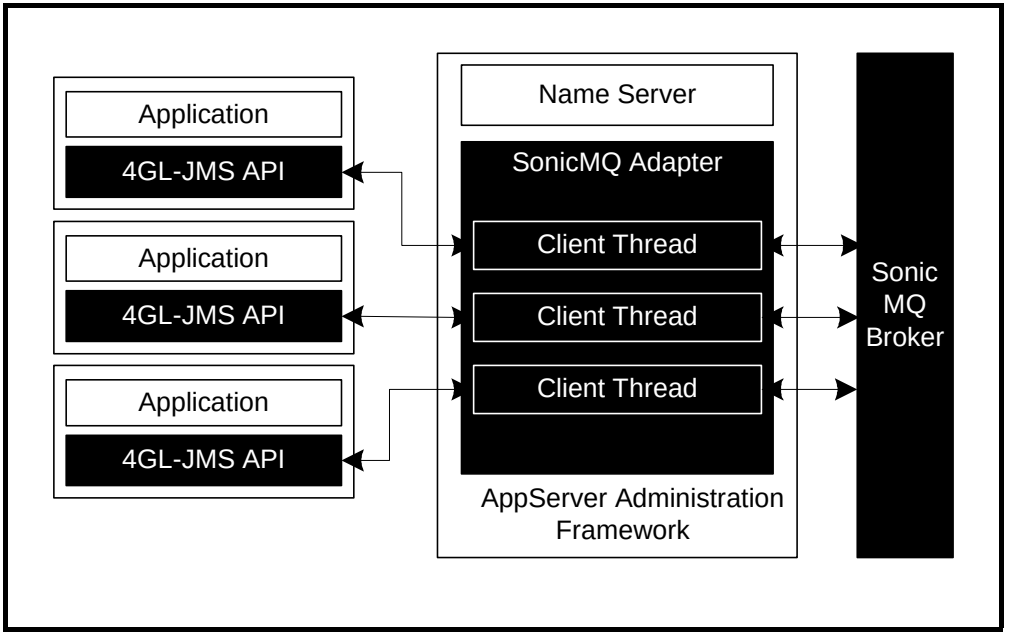


Figure 13–2: The SonicMQ Adapter Architecture

13.1.2 Requirements

The following components are required to access the SonicMQ broker from the 4GL:

- The SonicMQ product from Sonic Software Corporation
- The Progress SonicMQ Adapter

NOTE: Version 9.1D or later of the Progress SonicMQ Adapter requires Version 4 or later of the SonicMQ product.

- On the 4GL side, *progress-install-directory*/jms/*.r and *progress-install-directory*/jms/impl/*.r files. These files are installed with the SonicMQ Adapter product. When deploying to a system where the SonicMQ Adapter is not defined, copy the entire progress-install-directory/jms from a SonicMQ Adapter installation to the PROPATH for the deployed client.

The 4GL programmer should be familiar with:

- How to install and configure the SonicMQ *message provider* (a messaging system that implements the JMS interface)
- How to install and configure the SonicMQ Adapter
- The 4GL event model
- The basic concepts of the JMS model, although JMS programming is a plus for accomplishing complicated tasks

The 4GL programmer follows these general steps to send and receive messages:

1. Start two Progress processes: a NameServer instance and at least one SonicMQ Adapter instance.
2. Write 4GL programs that connect to the SonicMQ broker through the SonicMQ Adapter by creating a *progress-install-directory*/jms/pubsubsession.r persistent procedure or a *progress-install-directory*/jms/ptpsession.r persistent procedure.
3. Send and receive messages using the 4GL–JMS API implemented by those Session objects.

NOTE: A 4GL programmer does not have to write any Java or 4GL code on the server side. That code is supplied by Progress and installed with the SonicMQ Adapter.

13.1.3 4GL–JMS Object Model

The *4GL–JMS object model* is a model wherein a 4GL application interacts with a JMS messaging broker through 4GL persistent procedures that encapsulate the functionality of JMS sessions and JMS messages. The 4GL application interacts with the SonicMQ broker through internal procedures, which perform actions, and user-defined function calls to extract values. Asynchronous message arrival is handled by setting up 4GL message handling procedures. A *message handler* is a routine consisting of a procedure-handle, internal-procedure-name pair, written by the application and set in the Message Consumer object. When a message is received, the message handler is called automatically so the application can process the message.

The following sections describe Session, Message Consumer, and Message objects, how they correspond to 4GL procedures, and how to create them.

Session Objects

Progress supplies two session procedures, `jms/pubsubsession.p` and `jms/ptpsession.p`, for the 4GL application to interact with JMS. These procedures run persistently to represent a JMS session and its underlying connection:

- `jms/pubsubsession.p` — For Pub/Sub messaging
- `jms/ptpsession.p` — For PTP messaging

These session procedures implement internal procedures that return additional 4GL objects in the form of 4GL persistent procedure handles. A single 4GL session can have any number of `jms/pubsubsession.p` and `jms/ptpsession.p` procedures, each of which creates an underlying JMS connection.

Since the Pub/Sub and the PTP models use the same kind of messages and generally support similar strategies for message delivery, acknowledgment, and recovery, the `jms/pubsubsession.p` and `jms/ptpsession.p` procedures have similar methods. (For more information, see the "Connections and Sessions" section in this chapter.)

Message Consumer Object

A *Message Consumer* is a JMS messaging object that receives messages from a destination or receives asynchronous error messages. The `jms/pubsubsession.p` and `jms/ptpsession.p` procedures create the messageConsumer object to handle incoming messages (using the createMessageConsumer internal procedure). The 4GL application must set a message handler procedure in the messageConsumer object by implementing a 4GL internal procedure with a specific signature. To create the messageConsumer object, the application passes the name of that internal procedure, as well as a handle to the persistent procedure instance that defines that procedure, to the Message Consumer object. (For more information on the Message Consumer object, see the "Message Consumer Objects," "Subscribing To a Topic," and "Receiving Messages From a Queue" sections in this chapter.)

The SonicMQ Adapter integrates with 4GL event handling. Messages are processed by the message consumer when the 4GL is in a WAIT–FOR state or other IO blocking states. While in a WAIT–FOR or other IO blocking state, all other UI and non-UI events are handled normally. WAIT–FOR can be called explicitly or through the waitForMessages Session object method, which works the same for GUI, character, batch, AppServer, and WebSpeed applications.

SonicMQ provides several standard JMS message types, plus the XMLMessage type. Table 13–1 lists the JMS message types, the corresponding SonicMQ message types, and the Session object internal procedure that create those messages.

Table 13–1: Message Types

JMS Message Type	SonicMQ Message Type	Session Object Internal Procedure
TextMessage	TextMessage	createTextMessage
MapMessage	MapMessage	createMapMessage
StreamMessage	StreamMessage	createStreamMessage
BytesMessage	BytesMessage	createBytesMessage
ObjectMessage	ObjectMessage	Not supported by the 4GL–JMS API
Message	Message	createHeaderMessage
Not supported by JMS	MultiPartMessage	createMultipartMessage
Not supported by JMS	XMLMessage	createXMLMessage

NOTES:

- A header-only message is a basic SonicMQ message, which handles bodyless JMS messages (javax.jms.Message).
- The *XMLMessage* is a SonicMQ extension of the JMS TextMessage that supports the same methods as Text messages and provides for XML messaging.
- A multipart message can contain one or more of the following:
  - SonicMQ message
  - Character data
  - Binary data

13.1.4 Typical JMS Messaging Scenarios In a 4GL Application

The following sections describe various JMS messaging scenarios in a 4GL application. These scenarios include connecting to a SonicMQ broker, publishing and subscribing, sending messages to a queue, receiving messages from a queue, and deleting objects.

Connecting To a SonicMQ Broker For a Pub/Sub Session

Publish and Subscribe (Pub/Sub) is a domain of JMS messaging in which an application (or publisher) sends (or publishes) messages to topics. A topic is like a node in a content hierarchy. If an application is interested in one or more topics, the application can subscribe to them and whenever messages are published to those topics, the application receives them. The same application can be a publisher and a subscriber, and a single publisher can send a message to multiple subscribers.

The following general steps outline how a 4GL application connects to a SonicMQ broker for a Pub/Sub session:

1. The application runs `jms/pubsubsession.p` persistently.
2. The application sets connection parameters for SonicMQ.
3. The application calls beginSession to connect to the SonicMQ Adapter and the SonicMQ broker and start the JMS session.
4. The application uses the handle of `pubsubsession.p` to create and publish messages to topics and to subscribe to and receive messages from topics.
5. The application calls the deleteSession() internal procedure in `pubsubsession.p` to close the session and the underlying connection.

Publishing a Message To a Topic

The following general steps outline how a 4GL application publishes a message to a topic:

1. The application obtains a handle to `pubsubsession.p`.
2. The application creates a message by calling one of the create...Message procedures in the Session object.
3. The application populates the header fields, properties, and body of the message.
4. The application calls the publish() internal procedure in `pubsubsession.p` with the message handle and the name of a topic as input parameters.
5. If the application is not going to use the message after publishing, it deletes the message.

Subscribing To a Topic and Receiving Messages

The following general steps outline how a 4GL application subscribes to a topic and receives messages:

1. The application obtains a handle to `pubsubsession.p`.
2. The application creates a Message Consumer object by calling the createMessageConsumer internal procedure.
3. The application passes to the message consumer the name of an internal procedure and a handle to the procedure that contains that internal procedure. The internal procedure is the procedure that handles messages.
4. The application calls the subscribe() internal procedure in `pubsubsession.p` with the name of a topic and the message consumer handle as input parameters.
5. The application executes a WAIT–FOR statement (or a waitForMessages call) and processes incoming messages and other 4GL events.
6. The application deletes the messages after the application finishes using them.

Connecting To a SonicMQ Broker For a PTP Messaging Session

Point-to-Point (PTP) is a domain of JMS messaging in which an application, known as a sender, sends a message to a destination, called a queue. Another application, known as a receiver, receives that message from the queue. The same application can be a sender and a receiver, but a single message goes to only one receiver.

The following general steps outline how a 4GL application connects to a SonicMQ broker for a PTP session:

1. The application runs `jms/ptpsession.p` persistently and calls beginSession to start the JMS session.
2. The application uses the handle of `ptpsession.p` to create and send messages to a queue and to receive messages from a queue.
3. The application calls the deleteSession() internal procedure in `ptpsession.p` to close the session and the underlying connection.

Sending Messages To a Queue

The following general steps outline how a 4GL application sends a message to a queue:

1. The application obtains a handle to `ptpsession.p`.
2. The application creates a message by calling one of the create...Message procedures of the Session object.
3. The application populates the header fields, properties, and body of the message.
4. The application calls the sendToQueue internal procedure in `ptpsession.p` with the message handle and the name of a queue as input parameters.
5. The application can use the message more than once and then delete it.

Receiving Messages From a Queue

The following general steps outline how a 4GL application receives a message from a queue:

1. The application obtains a handle to `ptpsession.p`.
2. The application creates a Message Consumer object by calling the createMessageConsumer internal procedure.
3. The application calls the ReceiveFromQueue internal procedure in `ptpsession.p` with the name of a queue and the message consumer handle as input parameters.
4. The application executes a WAIT–FOR statement (or a waitForMessages call) and processes incoming messages and other 4GL events.
5. The application deletes the messages after it finishes using them.

Sending a Message and Receiving a Reply

The following general steps outline how a 4GL application sends a message and receives a reply:

1. The application obtains a handle to a session procedure (`pubsubsession.p` or `ptpsession.p`).
2. The application creates a message by running one of the create...Message internal procedures.
3. The application creates a Message Consumer object for handling replies by calling the createMessageConsumer internal procedure.
4. The application calls the requestReply() internal procedure in the Session object with the message handle, the name of a destination (a topic name in the `pubsubsession.p` case and a queue name in the `ptpsession.p` case), and the message consumer handle as input parameters.
5. The application executes a WAIT–FOR statement (or a waitForMessages call), which waits for the replies to arrive while processing other 4GL events.
6. The Message Consumer object handles the replies.
7. The application deletes reply messages after it finishes using them.

Deleting Objects

A 4GL application must explicitly delete 4GL objects after using them:

- The Session object, `pubsubsession.p` or `ptpsession.p`, implements the deleteSession internal procedure call.
- The Message object implements the deleteMessage internal procedure call.
- The Message Consumer object implements the deleteConsumer internal procedure call.

These calls delete the SonicMQ Adapter and server side resources as well as the 4GL persistent procedure of that instance.

13.1.5 Integrating With the Native 4GL Publish-and-Subscribe Mechanism

The JMS Pub/Sub model complements the Progress 4GL syntax for publish and subscribe (named events) for distributed applications. As shown in a Figure 13–3, 4GL programmers can write an application using the local Progress 4GL syntax for publish and subscribe and then make the local application distributed by adding local and remote gateway object modules. Using this model, a 4GL programmer can integrate the local application with the SonicMQ functionality without recompiling. Progress recommends this model, but does not provide specific software to implement it, except for the sample application files. (See the section "Gateway Sample Application" in Chapter 14, "Using the SonicMQ Adapter.")

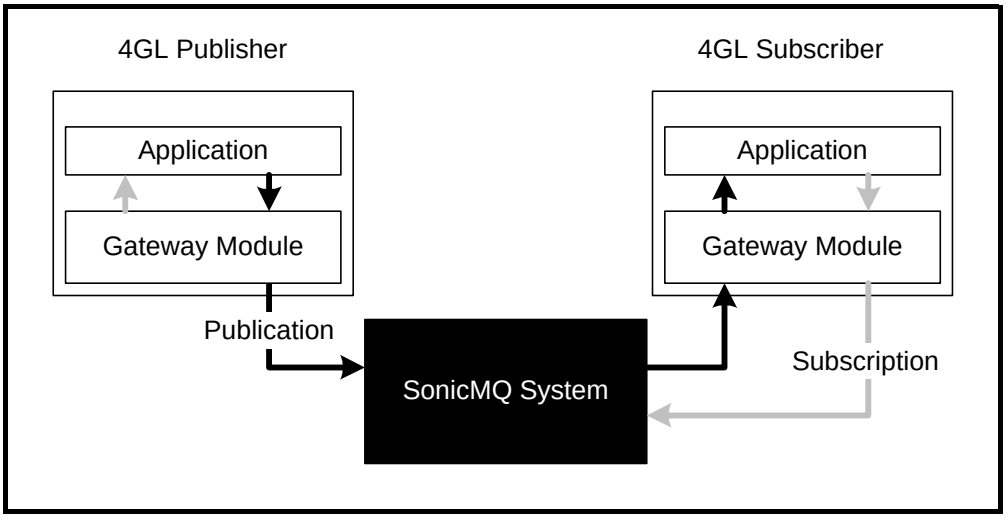


Figure 13–3: The Gateway Model

13.1.6 Building Scalable Server Architecture With PTP Queuing

A typical use of PTP messaging is to build a scalable and reliable server architecture. Both 4GL and non-4GL clients send requests to a JMS queue on a broker. 4GL servers remove messages from the queue, execute the requests, and reply to the clients. Reliability is provided since requests and replies do not get lost in case of a system failure. Scalability is achieved by providing an increasing number of 4GL servers as the number of clients and the rate of requests increases.

For an example, see the "Achieving Scalable Server Architecture With PTP Queuing" section in Chapter 14, "Using the SonicMQ Adapter."

13.2 Mapping JMS Objects To 4GL Objects

The full JMS Pub/Sub and PTP messaging models are accessible through the 4GL–JMS API. This API simplifies some of the most commonly used JMS features to minimize the code a 4GL programmer needs to write. Further, some Java–JMS API calls are modified to preserve the 4GL programming style. For example, the JMS method, getPropertyNames(), returns a Java Enumeration object, while its 4GL counterpart returns a comma-separated list of property names.

13.2.1 Connections and Sessions

In JMS, a Java client can create several sessions per connection. Each session is a single-threaded context for both sending and receiving messages. Since the 4GL is single-threaded, there is no compelling reason for multiple sessions per connection, nor for exposing the distinction between sessions and connections. In the context of the 4GL–JMS API, the combination of a session and a connection is called a session.

When more than one session per connection is required (for example, to send and receive messages concurrently), a second session is used implicitly on the SonicMQ Adapter broker, transparently to the 4GL programmer.

Creating a JMS Session

These are the general steps to create a JMS session in the 4GL:

1. Run `jms/pubsubsession.p` or `jms/ptpsession.p` persistently with the SonicMQ Adapter connection parameters as INPUT CHAR parameters.

2. Set JMS attributes and parameters by calling internal procedures in the session procedure (optional).

3. Start the actual JMS connection and session by calling the beginSession() internal session procedure.

NOTE: Session attributes cannot be modified after calling beginSession().

Creating a Session In a WebClient

A WebClient 4GL application connects to the AppServer and the SonicMQ Adapter through an HTTP server. To connect to the SonicMQ Adapter through HTTP, the input parameter to the `jms/ptpsession.p` or the `jms/pubsubsession.p` procedure call must be in the "–URL <url>" format and include the service name of the SonicMQ Adapter. If the URL format is used, the SonicMQ Adapter's service name is part of the URL and the setAdapterService 4GL–JMS API call is ignored. An example of a valid URL parameter is:

```
-URL http://host1:3099/uri?AppService=adapter.progress.jms
```

In sites where a proxy is used as part of the firewall configuration, the 4GL application might have to specify a proxy HTTP service in addition to specifying the URL connection parameter of the HTTP server. These parameters are specified as Progress startup parameters (–proxyHost and –proxyPort) and are used for all of a WebClient's AppServer and SonicMQ Adapter connections.

The other WebClient Session object methods are identical to those for a nonWebClient session.

For more information on connecting to the AppServer, see the chapter on the AppServer Internet Adapter in the *Progress Version 9 Product Update Bulletin*.

Deleting a JMS Session

An application calls deleteSession in the Session object to close and delete the session. This call terminates the underlying JMS connection and sessions, disconnects the 4GL client from the SonicMQ Adapter, deletes all the Message Consumer objects, and deletes the session's persistent procedure.

The deleteSession call does not delete the 4GL Message objects associated with the session; those messages remain for possible use with other sessions.

Creating Multiple Sessions

A 4GL application can create and use multiple Session objects concurrently.

13.2.2 Messages

The 4GL object model for messages is very similar to the Java object model. Each supported JMS message type has a 4GL counterpart, and, with some exceptions, each JMS message method is supported by a 4GL method with the same name. In particular, the getJMS... and setJMS... header methods are fully supported (See the *Java Message Service* specification and the *SonicMQ Programming Guide* for details.)

A Progress 4GL–JMS message can have either of two life cycles—one for creating a message and one for receiving a message.

The Progress 4GL–JMS message life cycle for creating a message has these general steps:

1. Create a message by running create...Message (for example, createXMLMessage) in the Session object.
2. Populate a message by running set... and write... for header and data information.
3. Send the message to a destination.
4. Run deleteMessage to delete the message.

The Progress 4GL–JMS message life cycle for receiving a message has these general steps:

1. Receive a message in a consumer object
2. Run get... and read... to extract header information and body data.
3. Run deleteMessage to delete the message.

Data Storage and Extraction Methods

An important difference between the 4GL model and the Java model is the difference in the names of the methods that extract data from a message:

- In Java, the name of the method determines the data type to be converted to. For example, readString() extracts a value such as an Integer data type from a StreamMessage and converts it to a String data type.
- In the 4GL, the equivalent function is readChar() to convert an Integer value to a 4GL CHARACTER data type.

When writing data to a message, an application uses the name of the data type to specify the Java data type in the message; the 4GL name is identical to the Java name. For example, Java uses the writeShort(short) method to write a number to a StreamMessage as short. The 4GL counterpart is the internal procedure, writeShort(value AS INTEGER).

Conversion between 4GL decimal values and Java double/float values follows the Java rules. (See the publications available at `http://www.java.sun.com.`)

Table 13–2 maps the 4GL data types to the JMS data types for data storage.

Table 13–2: Data Storage Table

4GL Data Type	JMS Data Type
LOGICAL	boolean
INTEGER	byte
INTEGER	short
INTEGER	int
DECIMAL	long
DECIMAL	float
DECIMAL	double
CHARACTER	String
A single CHARACTER	char
RAW	bytes array
MEMPTR	bytes array (only with BytesMessage)

Table 13–3 maps the available conversions from JMS data types to 4GL data types for data extraction.

Table 13–3: JMS and 4GL Data Types For Extracting Data

JMS Data Type	4GL Data Type
boolean	LOGICAL or CHARACTER
byte	INTEGER, DECIMAL, or CHARACTER
short	INTEGER, DECIMAL, or CHARACTER
int	INTEGER, DECIMAL, or CHARACTER
long	DECIMAL or CHARACTER
float	DECIMAL or CHARACTER
double	DECIMAL or CHARACTER
String	CHARACTER
char	CHARACTER
bytes array	RAW or MEMPTR (MEMPTR is available only with BytesMessage)

Message Handles

As in Java, a 4GL application obtains a handle to a message by creating it or receiving the message handle as an INPUT parameter of the message handler procedure. The 4GL application creates a message of a particular type by making a create...Message call (for example, createTextMessage) in the Session object. Messages are received by the message handler that was installed in a Message Consumer object. (For more information, see the "Message Consumer Object," "Subscribing To a Topic," and "Receiving Messages From a Queue" sections in this chapter.)

Message Typing

Unlike Java, 4GL Message objects are not typed. Therefore, the 4GL application must call the getMessageType function (returns CHAR) if it does not know what message type it expects to receive, so that it can make the correct function calls to extract the data.

Header Methods

All message types have the same header information and support the same methods, mostly for setting and getting header information. This follows the Java model in which all message types extend the basic Message interface. (For more information, see the "Message Objects" section in this chapter.)

Deleting Messages

Unlike Java, which uses garbage collection, with the 4GL–JMS API, messages that are not used must be explicitly deleted using the deleteMessage internal procedure (supported by all message types).

TextMessage

A *TextMessage* is a message type whose body contains text data. To send and receive a TextMessage over 32K, the 4GL–JMS API extends the JMS API with methods to concatenate text segments. As with Java–JMS, text data is extracted and stored in a message by the getText and setText methods for any TextMessage less than 32K.

To send and receive a TextMessage that exceeds the 32K limit, the 4GL–JMS API extends the JMS API with the appendText and getTextSegment methods. With multiple appendText calls, a 4GL client can create a TextMessage up to the limit of the JMS server. The JMS non-4GL client receives one TextMessage consisting of the concatenation of all the text segments.

To allow the 4GL client to receive messages larger then 32K, the server segments the received TextMessage into text segments of 8K (8192) or fewer characters. An application can then use multiple getTextSegment methods to retrieve all the segments. If getText is called, the 4GL–JMS API returns all the text. If the TextMessage is too large for the 4GL interpreter to handle, a run-time error occurs.

For example, if the message value is UNKNOWN, or "", or a String of 5,000 characters, an application can use one getText call (or one getTextSegment call). If the message size is 16,400 characters, the first two getTextSegment calls return 8192 characters each, and the last getTextSegment call returns 16 characters. An application can use the getCharCount call to get the total number of characters in a message.

The endOfStream TextMessage function returns true when all of the segments are retrieved (the number of getTextSegment calls matches the number of segments). The setText call implicitly calls clearBody before setting the new text. The reset and getText calls transfer the message from write-only to read-only mode and position the message cursor before the first segment. (For more information, see the "Read-only and Write-only Modes" section in this chapter.)

NOTE: The 8K segment size is guaranteed. A 4GL application does not have to use the endOfStream function for messages smaller then 8K, since there is only one segment. For information about code-page conversions and text-size limits, see the "Internationalization Considerations" section in Chapter 14, "Using the SonicMQ Adapter."

XMLMessage

The 4GL–JMS API supports SonicMQ's XMLMessage. As with SonicMQ, *XMLMessage* is an extension of a JMS TextMessage. XMLMessage supports the same methods as TextMessage.

XML messages can be used in conjunction with the 4GL XML parser:

- **Incoming messages** — Move the XML text into a MEMPTR and load into the X–DOC object.
- **Outgoing messages** — Save the XML text into a MEMPTR and set in the message.

It is important to consider the code page of XML messages. Theoretically, XML documents can be encoded using any code page. However, each XML parser supports some or all code pages, and XML parsers differ in the code-page conversions they can do. (A *code page* is table that maps each character on it to a unique numeric value.)

With the 4GL–JMS API, the conversion rules are straightforward. The text stored in an XML message by the 4GL application is expected to be encoded in the internal code page of the 4GL client (the –cpinternal startup parameter).

The 4GL–JMS implementation automatically converts the text to Unicode when a SonicMQ XML message is created. *Unicode* is an encoding format that provides a unique number for every character, no matter which platform, program, or language. The 4GL–JMS implementation also converts the Unicode text received in XML messages to the internal code page of the 4GL client when the text is extracted. For more information, see the "XML Code-page Encoding" section in this chapter.

StreamMessage

A *StreamMessage* is message whose body contains data that is written and read as a stream of basic data types; it is filled and read sequentially. There is a difference in the way a StreamMessage is read with the Java–JMS model and the 4GL–JMS model:

- With Java–JMS, the read... call moves the cursor to the next item.
- With the 4GL–JMS model, the moveToNext method moves the cursor to the next item and returns the data type of the item. Then the read... call retrieves the data.

A separate call to retrieve the data type is not required in the Java–JMS model since the data type can be derived from the object type returned by the readObject Java method.

BytesMessage

A *BytesMessage* sends a stream of uninterpreted bytes. This message type allows passing data "as is" without any interpretation by the 4GL–JMS API or the JMS server. For example, a BytesMessage can pass an XML document encoded in a code page that does not match the 4GL client's code page. (For more information, see the "XML Code-page Encoding" section in this chapter. For an example, see the "Publishing, Subscribing, and Receiving an XML Document In a BytesMessage" section in Chapter 14, "Using the SonicMQ Adapter.")

An application writes data to a BytesMessage using RAW or MEMPTR variables with writeBytesFromRaw or setMemptr and reads data with g readBytesToRaw or getMemptr. (For an example, see the "Publishing, Subscribing, and Receiving the Customer Table In a StreamMessage" section in Chapter 14, "Using the SonicMQ Adapter."

As with a TextMessage, the 4GL–JMS API accommodates messages larger than 32K by allowing multiple readBytesFromRaw and writeBytesToRaw calls. (For more information, see the "TextMessage" section in this chapter.)

MultipartMessage

A *MultipartMessage* can contain one or more of the following:

- SonicMQ message
- Character data
- Bytes data

To create a multipart message, use the createMultipartMessage method. For a reference entry, see the createMultipartMessage section in Appendix C, "4GL–JMS API Reference." For an example, see the MultipartMessage section in this chapter.

Java Object Messages

Java Object messages are not supported by the 4GL–JMS API. If a Java Object message is received on behalf of a 4GL client, the client's asynchronous error handler receives a TextMessage with the header of the Java Object message and a text body with the string, "ObjectMessage: Not Supported." (For more information, see the "Error and Condition Handling" section in this chapter.)

Read-only and Write-only Modes

As in Java–JMS, the StreamMessage, TextMessage, XMLMessage, and BytesMessage are created in write-only mode. In write-only mode, an application can use only data-setting methods, not data-extraction methods. The reset call puts the cursor before the first item of the message and transfers it to read-only mode. Note that the publish, sendToQueue, and requestReply methods call reset implicitly. The message is received by the receiver in reset mode. The clearBody call transfers the message to write-only mode.

Note that read-only and write-only refer to the body of the message, not its header. Read-only and write-only modes do not apply to Header messages, since they lack a body.

Unlike in Java–JMS, a MapMessage in the 4GL–JMS implementation is always in read/write mode; there is no read-only or write-only mode for a MapMessage.

Reset has no effect when called in Map and Header messages.

clearBody and clearProperties

The clearBody and clearProperties methods are supported by all message types:

- The clearBody method deletes all the data from the message body.
- The clearProperties method deletes all the header properties (but not the JMS predefined header fields).

Message Size Limits

There is no limit to the 4GL message size. SonicMQ does not have a hard-coded maximum message size; the largest tested message is 1MB. The 4GL imposes a 32K limit on each item of a StreamMessage or a MapMessage. For more information about code-page conversions and text size limits, see the "Internationalization Considerations" section in Chapter 14, "Using the SonicMQ Adapter."

When using very large messages (above 1MB), you might need to modify the JVM's memory limit values in the jvmArgs property of the SonicMQ Adapter broker (in the `ubroker.properties` file). For example, if the SonicMQ Adapter fails with an OutofMemory error in the log, you should modify:

```
-mx, -ss, and -oss
```

This example specifies 40MB for the memory heap, 8MB for the stack, and 8MB for the stack object memory:

```
-mx40m -ss8m -oss8m
```

Limit To the Number Of Messages

The value of the 4GL global variable, JMS–MAXIMUM–MESSAGES, determines the maximum number of Message objects in a 4GL session. An application must delete unused messages using the deleteMessage method of the Message object to avoid exceeding the JMS–MAXIMUM–MESSAGES value. The default value of JMS–MAXIMUM–MESSAGES is 50. An application can change the value of JMS–MAXIMUM–MESSAGES by declaring it in the 4GL application. For example, to change the value of JMS–MAXIMUM–MESSAGES from 50 to 100, the main 4GL procedure of the application must include the following definition:

```
DEFINE NEW GLOBAL SHARED VARIABLE JMS-MAXIMUM-MESSAGES AS INT INIT 100.
```

13.2.3 Publishing a Message To a Topic

In the Pub/Sub domain, applications publish messages to topics.

To publish a message with Java–JMS, an application obtains a handle to a topic object and creates a publisher object and then uses the publisher to publish messages.

With the 4GL–JMS implementation, an application publishes messages through the publish internal procedure of the `pubsubsession.p` object. The topic name is specified as a CHAR INPUT parameter. Other publish parameters (such as persistency, timeToLive, and priority) can be set in the `pubsubsession.p` object as a default for all the published messages or can be set at each publish call.

NOTE: If a message being published to a topic has a delivery mode of DISCARDABLE and the destination topic queue is full, the message is automatically discarded.

13.2.4 Sending a Message To a Queue

In the PTP domain, applications send messages to a queue.

To send a message to a queue with Java–JMS, an application obtains a handle to a queue object, creates a Queue Sender object, and uses the Queue Sender to send messages.

Sending a message to a queue with the 4GL–JMS API involves these general steps:

1. The application calls the sendToQueue internal procedure in the `ptpsession.p` object.

2. The application specifies the queue name as a CHAR INPUT parameter.

   The application can set other sending parameters (such as persistency, timeToLive, and priority) in the `ptpsession.p` object as a default for all the messages it sends, or it can set these parameters at each sendToQueue call.

13.2.5 Subscribing To a Topic

In the Pub/Sub domain, applications subscribe to topics of interest.

To subscribe to a topic with Java–JMS, an application obtains as handle to a topic object, creates a subscriber object, and installs a message-handling routine in the subscriber object.

To subscribe to a topic with the 4GL–JMS API, an application implements a message handler routine for handing the incoming messages and a Message Consumer object that contains the message handler and provides context to the application when it processes messages.

Creating a Message Handler

A message-handler routine must implement a specific signature with two input parameters (the incoming message and the containing messageConsumer object) and one output parameter (for the reply message). For information about the signature of a message handler, see the "Message Handler" section in this chapter.

Creating a Message Consumer

An application creates a messageConsumer object by calling createMessageConsumer in the Session object. The input to createMessageConsumer is the handle and the name of the message handler.

In addition, the application can set more context information in the Message Consumer object. For example, an application can use the setApplicationContext procedure in the Message Consumer object to set a handle (typically a procedure handle) that the message handler (which calls getApplicationContext) can use to receive context information and to communicate message processing results to the rest of the application.

If the setReuseMessage message consumer method is called, the Message Consumer object will reuse the same Message object for all the messages it receives, provided that the message was not deleted by the application. Using setReuseMessage improves performance since message creation is relatively expensive.

Subscribing

The application calls the subscribe procedure in `pubsubsession.p` with the topic name and a handle to the messageConsumer object. For the 4GL application to subscribe to a durable subscription, it must provide the name of the subscription as well. A *durable subscription* is a subscription registered with the SonicMQ broker with a unique identity so the broker retains the subscription's messages until they are received by the subscription or until they expire. The application can pass a JMS properties selector expression to the subscribe call to specify which messages the subscriber wants to receive. The application can also specify whether or not it wants to receive its own published messages.

13.2.6 Receiving Messages From a Queue

In the PTP domain, applications receive messages from a queue.

To receive messages from a queue with Java–JMS, an application obtains a handle to a queue object, creates a message receiver object, and installs a message-handling routine in the message receiver object.

With the 4GL–JMS API, the general steps to receive messages from a queue are similar, except:

1. The application implements a message-handler routine to handle the incoming messages.
2. The application creates a messageConsumer object that contains the message handler and provides context to the application when it processes the messages.

Creating a Message-handler Routine

A message-handler routine must implement a specific signature with two input parameters (the incoming message and the containing messageConsumer object) and one output parameter (for the reply message). For information about the signature of a message handler, see the "Message Handler" section in this chapter.

Creating a Message Consumer

The application creates a messageConsumer object by calling createMessageConsumer in the Session object. The input to createMessageConsumer is the handle and the name of the message handler.

In addition, the application can set more context information in the messageConsumer object. For example, the setApplicationContext procedure in messageConsumer is used by the 4GL application to set a handle (typically a procedure handle) that can be used by the message handler (which calls getApplicationContext) to receive context information and to communicate message processing results to the rest of the application.

Receiving Messages From the Queue

Next, the application calls the receiveFromQueue procedure in `ptpsession.p` with the queue name and a handle to the messageConsumer object. The application can pass a JMS properties selector expression to the receiveFromQueue call to specify which messages the receiver wants to receive from the queue.

13.2.7 Message-reception Issues

The following sections discuss several message-reception issues in the Pub/Sub and PTP domains.

Stopping and Starting Message Reception

To actually start receiving messages, the 4GL application must call the startReceiveMessages procedure in the Session object. One call to startReceiveMessages is sufficient for the session. The application typically calls the startReceiveMessages procedure after subscribing to all topics of interest (in the Pub/Sub domain) or registering Message Consumer objects with the queues of interest (in the PTP domain).

The application can also call stopReceiveMessages to temporarily stop the reception of messages. The application can then call the startReceiveMessages procedure again to resume message reception.

In the Pub/Sub domain, calling stopReceiveMessages does not cancel any existing subscription, but if the subscription is not durable, messages published while reception is stopped are not delivered.

In the PTP domain, the messages are queued while the client is in a stopReceiveMessages state and are delivered to the client after startReceiveMessages is called again.

Stopping the reception of messages is recommended when an application is not going to process messages for a while. If stopReceiveMessages is not called, messages are queued in a 4GL–JMS temporary queue on the server side, potentially consuming a large amount of memory.

NOTE: After calling stopReceiveMessages, the 4GL client might receive one message that was sent from the server prior to the execution of the stopReceiveMessages call.

Message Consumer Scope

A messageConsumer object can be used to handle only one subscription (in the Pub/Sub domain) or receive messages from one queue (in the PTP domain). When its deleteConsumer procedure is called, message reception is canceled and the messageConsumer object is deleted.

NOTE: To delete a durable subscription (in the Pub/Sub domain), the cancelDurableSubscription() procedure in `pubsubsession.p` must be called as well, since deleteConsumer only suspends the subscription in the current session. There is no equivalent to a durable subscription in the PTP domain.
It is an error to call cancelDurableSubscription() if there is an active message consumer for that subscription. First call deleteConsumer to delete the message consumer.

When a messageConsumer object is used for receiving replies through the requestReply call, it can be used many times; there is no need to create one for every call. The deleteSession call deletes all messageConsumer objects for that session.

4GL Interpreter Message Processing States

A 4GL application receives and processes messages when it is in an I/O blocking state. The same rules that determine when ASYNC completion procedures are fired also determine when message handlers are called. The 4GL application should typically use the WAIT–FOR statement or the waitForMessages API session call, for processing messages as well as for other events.

The waitForMessages call is a convenient way to write message-handling code that is independent of the environment in which the 4GL application is executed (GUI, CHUI, batch, AppServer, or WebSpeed). It allows the application to specify when to stop waiting and it processes all events that occur while waiting, including user-interface events and ASYNC call events.

The waitForMessages call takes three input parameters: a procedure handle, a name of a user-defined function in the procedure that returns LOGICAL, and a timeOut INTEGER. The waitForMessages call waits and processes events as long as the user-defined function returns true and there is no period of more than timeOut seconds in which no messages are received.

The user-defined function is evaluated by the 4GL–JMS API after the message handler is executed. Typically, the 4GL application should have logic for changing the return value of the function in the message handler.

Synchronous Message Reception

The 4GL does not explicitly support receiving messages synchronously. The same effect can be achieved by using the WAIT–FOR statement (or a waitForMessages call), which waits for a user-defined event. When the desired message is received, the message handler can trigger the termination of the WAIT–FOR statement (or the waitForMessages call) by, for example, applying the user-defined event the WAIT–FOR statement is waiting for.

13.2.8 Reply Mechanisms

This section applies to both the Pub/Sub and the PTP domains. In Java–JMS, there is no built-in mechanism for replies. It is the responsibility of the application to designate a destination (typically a temporary destination) for replies. It is also the responsibility of the application to send this Destination object to the receiver (typically through the replyTo message header field). (A *message header* is a group of fields in a message containing values to identify and route messages.) The receiver must extract the reply destination from the message and follow the normal publish (or send) steps to reply.

The 4GL–JMS API simplifies that process for the sender who wants to receive a reply as well as for the receiver who wants to reply:

- **Sender** — The 4GL–JMS API requestReply method can publish messages the same way as the publish method, or send messages to a queue the same way as the sendToQueue method. In addition, a messageConsumer object for replies is passed to it as an input parameter (see the "Subscribing To a Topic" and "Receiving Messages From a Queue" sections for information about message reception). The 4GL–JMS implementation automatically routes all the replies to that messageConsumer object.

- **Receiver** — To reply, the message receiver returns a reply message handle as an output parameter in the message-handler routine. The application can call the setReplyAutoDelete procedure in the messageConsumer object to automatically delete the replies after sending them.

An application can also publish a reply message (or send it to a queue) by extracting the name of the reply destination by calling getJMSReplyTo and then calling publish (or sendToQueue) directly.

NOTE: If the replyTo destination is a temporary destination, an application must send a reply before deleting the original message. (See the *Java Message Service* specification and the *SonicMQ Programming Guide* for information on temporary destinations.) Deleting the original message tells the 4GL–JMS implementation that the replyTo temporary destination is not going to be used anymore.

By default, the type of the ReplyTo destination matches the type of the origin of the message:

- If the message was created by a `pubsubsession` object, the value of the ReplyTo field is considered a topic name.
- If the message was created by a `ptpsession` object, the value of the ReplyTo field is considered a queue name.

However, it is legal to designate a reply queue for a published message and to designate a reply topic for messages received from a queue. To accommodate this, the 4GL–JMS API supports the setReplyToDestinationType procedure and the getReplyToDestinationType function. The values that an application can set or get are “topic” and “queue.” The setReplyToDestinationType procedure must be called when the 4GL application calls setJMSReplyTo with a destination from a different domain than the session. The getReplyToDestinationType function must be called when the 4GL application receives a message and wants to reply to it, but is not certain about the ReplyTo domain.

Using a Second Session For Replying

There are scenarios where the 4GL application might need to reply through a session different from the one that received the original message. For example, the publisher of the message might like to designate a queue, rather than a topic, for receiving the replies. The publisher would set the JMSReplyTo header field to the name of the queue and then publish the message. The receiver would receive the message through a Pub/Sub session, extract the queue from the JMSReplyTo header field, and send the reply to that queue using `ptpsession.p`. Then, the original publisher would use `ptpsession.p` to receive the reply.

The 4GL–JMS API supports this functionality with these limitations:

- The JMSReplyTo header field cannot be set with the name of a temporary destination that came from a session different from the one used to send the message.
- If the 4GL application sets the JMSReplyTo with a destination domain different from the session that will be used to send the message, it must use the Java Naming and Directory Interface (JNDI) (or other administered object storage) name of that object. (See the “Finding Administered Objects In JNDI Or Proprietary Directories” section in Chapter 14, “Using the SonicMQ Adapter.”)
- An automatic reply (through the output reply parameter of the message-handling routine) does not work if the domain of the JMSReplyTo header field is different than the domain of the session. The application must send the reply explicitly through a second session.

13.2.9 Queue Browsing

The PTP model supports *queue browsing*, a mechanism for an application to view the content of messages in a queue without actually receiving (consuming) the messages. The 4GL–JMS API supports queue browsing through the browseQueue method in the ptpsession object. The browseQueue method receives the name of the queue, a selector expression, and a Message Consumer object as input parameters.

The messages can be handled by the message handler the same way as messages coming from a receiveFromQueue call, but they are not acknowledged and are not subject to the transactional context of the session. (See the *Java Message Service* specification and the *SonicMQ Programming Guide* for details on queue browsing.)

13.2.10 Message Acknowledgment and Recovery

A client sends an *acknowledgement* to tell the SonicMQ broker that the client received and processed a message and does not need to receive that message again. The acknowledgment of a message guarantees that the message and all previous messages are not delivered again to that session.

The following sections describe automatic message acknowledgment, preventing message acknowledgment, and message recovery.

Automatic Message Acknowledgement

With the 4GL–JMS API, an incoming message is acknowledged automatically when the message handler finishes execution. It is sent on the request for the next message, which helps performance.

If there is a client or communication failure between the time the message handler finishes execution and the time the 4GL–JMS implementation sends the acknowledgment, the message might be redelivered (according to the JMS message redelivery rules). An application can use a transacted session to avoid this message redelivery problem.

Unlike Java–JMS, the 4GL–JMS API does not support the explicit acknowledgment of messages or the “lazy” acknowledgment of messages (the CLIENT_ACKNOWLEDGE and DUPS_OK_ACKNOWLEDGE JMS modes).

Preventing Message Acknowledgment

A 4GL application can explicitly prevent the acknowledgment of a message by calling the setNoAcknowledge method of the messageConsumer. (The messageConsumer object is passed as a parameter to the message-handler procedure.) The setNoAcknowledge method is typically used when the application wants to receive the same message again due to some error in processing it or when it needs to acknowledge a group of messages by acknowledging only the last message in the group.

add para on SINGLE_MESSAGE_ACKNOWLEDGE mode here

Single Message Acknowledgement

Normally, a 4GL client application automatically acknowledges a message when the message handler procedure completes. But in SINGLE_MESSAGE_ACKNOWLEDGE mode, each message requires its own acknowledgement, and if you choose to not acknowledge a message, the message is never acknowledged.

To turn on SINGLE_MESSAGE_ACKNOWLEDGE mode, a 4GL client application calls the setSingleMessageAcknowledgement() method of the session handle with the input parameter set to TRUE. To turn off this mode, the application calls the same method with the input parameter set to FALSE.

Message Recovery

If an application wants to again receive all the messages that were not acknowledged, it can call the recover procedure in the Session object. If recover is called on a stopped session (stopReceiveMessages was called), the session is recovered and message delivery is restarted.

13.2.11 Transacted Sessions

A *transacted session* allows an application to send or receive groups of messages as one atomic operation:

- A session that is transacted for sending guarantees that either all the messages in a group are sent or none are sent.
- A session that is transacted for receiving guarantees that a group of received messages are acknowledged only after all the messages of that group are successfully processed.

The application controls whether sending is transacted, message receiving is transacted, or both. The typical Java–JMS transacted application uses two sessions, one for transacted sending and one for transacted receiving. The 4GL–JMS implementation uses two JMS sessions behind the scenes, but at the 4GL API level, there is only one Session object.

An application can call the setTransactedSend, the setTransactedReceive procedure, or both, in the Session object to make the session transacted. An application calls both procedures to make a session transacted for sending and receiving.

A transacted session (for sending, receiving, or both) is constantly in a transaction mode. When a transaction is committed or rolled back, a new one is automatically started.

Transacted Sending

When an application calls commitSend in a Session object, all messages that were published (or sent to queue) up to that point with the current transaction, are sent. When an application calls rollbackSend in a Session object, all messages that were published up to that point (or sent to queue) with the current transaction, are discarded.

Transacted Receiving

When an application calls commitReceive in a Session object, all messages that were received up to that point with the current transaction are acknowledged. When an application calls rollbackReceive in a Session object, all messages that were received up to that point with the current transaction are re-received (the same effect as calling recover with a nontransacted session).

No Recover and setNoAcknowledge Calls

Since message acknowledgment and recovery are handled automatically in a transacted session, it is an error to call the recover and setNoAcknowledge methods in a session that is transacted for receiving.

4GL Transactions and JMS Transacted Sessions

4GL transactions and JMS transactions are not integrated. For example, a DO TRANSACTION block might be rolled back, while the JMS calls inside the transaction block are committed. The 4GL application must synchronize between 4GL transactions and JMS transactions.

13.2.12 Error and Condition Handling

This section provides information about error and condition handling with the 4GL–JMS API.

Categories Of Errors and Conditions

From the point of view of the 4GL programmer, there are two types of errors and conditions, programming errors and run-time conditions.

- A programming error is an erroneous sequence of calls to the 4GL–JMS API or the calling of the API with invalid parameters. Typically, programming errors should not occur in a deployed application.

  An example of a programming error is an attempt by the application to make a TextMessage call in a StreamMessage. Programming errors should be tracked down and fixed at development time. The primary source of information for that phase is the 4GL–JMS API. (See Appendix C, "4GL–JMS API Reference.")

- A run-time condition is an event that disturbs the normal flow of the application and might happen in a deployed application. The 4GL programmer should try to handle it programmatically.

  Examples of run-time conditions include attempting to connect to a JMS server that is not currently running and trying to subscribe to a topic without the proper authorization. The primary source for understanding and programmatically handling run-time conditions is the *Progress SonicMQ JavaDoc* (in `\docs\api` under the SonicMQ installation directory).

A second category for classifying errors and conditions is whether or not the problem is reported by the 4GL–JMS implementation synchronously or asynchronously:

- A problem is reported synchronously if it occurs and is detected while the 4GL application is executing a 4GL–JMS API call.

- A problem is reported asynchronously when it comes from the asynchronous error reporting system of the JMS server (OnException Events) or from the 4GL–JMS mechanism that delivers messages asynchronously to the 4GL client.

A synchronous problem is reported by an API internal procedure by returning:

```
RETURN ERROR <error message>.
```

A 4GL API function reports a synchronous problem by returning an UNKNOWN value. These problems could be either programming errors or runtime conditions.

NOTE: Some synchronous programming errors are not detected by the 4GL–JMS but rather by the 4GL interpreter. For example, an attempt to call setText in a StreamMessage causes error 6456:

```
Procedure message.p has no entry point for setText. (6456)
```

An asynchronous problem is reported as a TextMessage and handled by the error-handler procedure. The error-handler procedure is set by the application using the setErrorHandler procedure in the Session object.

Programming errors are usually reported synchronously. Run-time conditions are reported either synchronously or asynchronously.

Asynchronous Conditions

Typically, problems reported asynchronously are run-time conditions. An example of an asynchronous problem is the failure of the SonicMQ broker or the failure of communication between the SonicMQ Adapter and the SonicMQ broker. (See the "Installing an Error Handler To Handle an Asynchronous Error" section in Chapter 14, "Using the SonicMQ Adapter.") Another example of an asynchronous condition is the failure to send an automatic reply. (The message handler is set with a reply message, but the SonicMQ server fails to send the reply.)

Asynchronous conditions should be handled programmatically by creating a messageConsumer object and passing it to the setErrorHandler procedure in the Session object. Asynchronous conditions are always reported in a TextMessage with several possible CHAR message properties in the message header: exception, errorCode, linkedException–1, linkedException–2… linkedException–N (where N is a number of additional exceptions linked to the main exception). Use the getPropertyNames function to get a list of properties in the error message header. (See the example in the "Installing an Error Handler To Handle an Asynchronous Error" section in Chapter 14, "Using the SonicMQ Adapter.")

If an application does not set an error handler, a default error handler displays the error message and the properties in alert boxes.

NOTE: An application must call beginSession before creating the error-handling messageConsumer object and calling setErrorHandler.

Synchronous Errors and Conditions

Errors are reported synchronously when something goes wrong at a method call; a 4GL ERROR condition is raised. Examples include trying to publish to an unauthorized topic or trying to receive from a nonexistent queue.

To report a problem synchronously, the 4GL–JMS API internal procedure calls:

```
RETURN ERROR <error message>
```

That call raises an error condition at the caller. The caller can use regular 4GL techniques to handle the error: a NO–ERROR phrase or an ON ERROR block, coupled with checking the RETURN–VALUE value to obtain the error message. If an application uses the NO–ERROR phrase, it must check the STATUS–ERROR:error flag to determine whether a problem has occurred.

By default, every synchronous error or condition is displayed by the 4GL–JMS API, which calls:

```
MESSAGE <error-message> VIEW-AS ALERT-BOX.
```

This allows a quick analysis and resolution of the problem at development time.

At deployment time, however, the application developer might want to handle problems programmatically and prevent the message from being displayed on the screen. This can be achieved by calling the setNoErrorDisplay(true) procedure in the Session object. Both setNoErrorDisplay(true) and setNoErrorDisplay(false) can be called several times during the life of a session.

NOTE: Message objects inherit the display/noDisplay property from the session that created them. However, after a message is created, it is independent from the session. The setNoErrorDisplay procedure must be called in the Message object itself to change this property.

Run-time Conditions

The typical run-time condition is generated by the Java–JMS code on the server. In that case, the format of the error message obtained from the RETURN–VALUE is:

```
<java-exception>:<error-message>.
```

The 4GL programmer can look up the type of exceptions thrown by SonicMQ and handle some of those programmatically. The most typical run-time conditions are connection and authorization failures.

Connection and Communication Failures

The most common run-time condition is a connection failure. A connection to the SonicMQ Adapter and the JMS server is created by the beginSession call. A connection failure is always reported synchronously by beginSession, which calls:

```
RETURN ERROR <error message>.
```

The connection can fail because of a failure to connect to the SonicMQ Adapter or a failure to connect to the JMS server. If the connection fails because of failing to connect to the JMS server, the format of the error message is:

```
<java-exception>:<error-message>.
```

A communication failure after establishing a successful connection might be detected:

- Synchronously, for example, when the application is trying to publish a message
- Asynchronously through the error handler

In some cases, it takes a long time for the underlying communication system to detect that there is a problem (possibly several minutes until the timeout mechanism triggers a communication failure event). Use the setPingInterval (N) session method to instruct the SonicMQ Adapter to actively ping the SonicMQ broker every N seconds to detect a communication failure more quickly. The setPingInterval functionality is a SonicMQ extension. A pingInterval value can also be specified in the srvrStartupParam property of the SonicMQ Adapter as a default for all clients.

Message Handler Errors and Conditions

A message handling procedure is an arbitrary 4GL program, and the 4GL programmer is free to handle problems that occur during the processing of a message using any 4GL technique.

However, the following issues and limitations must be considered:

- Message handlers should handle ERROR, STOP, and QUIT conditions and not propagate them. An unhandled condition is considered a programming error.
- Since the message handler returns control to the 4GL–JMS implementation and the message handler cannot raise a condition, there must to be a mechanism to allow the message handler to communicate problems to the rest of the 4GL application. This can be done by passing a 4GL procedure handle to the messageConsumer object using the setApplicationContext call. The message-handler procedure can obtain the procedure handle by calling the getApplicationContext procedure in the messageConsumer object and then making internal procedure calls in it.
- As mentioned in the "Message Acknowledgment and Recovery" section, the message handler can call the setNoAcknowledge method of the message consumer to prevent the message from being acknowledged in a session that is not transacted for receiving.
- Calling WAIT–FOR is allowed inside a message handler, but no further messages from that Session object are received until the message handler returns.
- The following recursive calls from the message handler into the 4GL–JMS API of the same Session object are considered a programming error: deleteSession, deleteConsumer, and recover. There are no restrictions on calling these API entries of another Session object.

Interrupts

An interrupt (**CTRL-C** on UNIX and **CTRL-BREAK** on Microsoft platforms) while a 4GL–JMS call is executing can cause either a 4GL STOP condition or an ERROR condition returned from the call, depending on the exact timing. The 4GL–JMS implementation guarantees that partial messages will not be sent or received as a result of an interrupt.

SonicMQ Adapter Failure

If communication with the SonicMQ Adapter is lost, or if the SonicMQ Adapter shuts down while the 4GL client is performing a WAIT–FOR (or waitForMessages) statement, a 4GL STOP condition is raised.

If communication with the SonicMQ Adapter is lost, or if the SonicMQ Adapter shuts down while the 4GL–JMS implementation is actively trying to communicate to it (for example, when the 4GL application calls publish or subscribe), an ERROR or STOP condition is raised, depending on the exact point where the failure was discovered.

13.3 Programming With the 4GL–JMS API

The following sections provide information on the 4GL–JMS API. This information is presented for the three objects:

- Session Objects
- Message Consumer Objects
- Message Objects

For an alphabetical API reference, see Appendix C, "4GL–JMS API Reference."

13.3.1 Session Objects

The Session objects, `jms/pubsubsession.p` and `jms/ptpsession.p`, encapsulate JMS connections and sessions.

The life cycle of a Session object consists of these general stages:

1. The application creates a session, but is not yet connected to the SonicMQ Adapter.
2. The application sets connection and session attributes.
3. The application runs beginSession to connect to the SonicMQ Adapter.
4. The application uses session methods to publish and subscribe or send and receive messages from a queue.
5. When the application finishes, it calls deleteSession to delete the connection from the application to the SonicMQ Adapter, delete the connection from the SonicMQ Adapter to SonicMQ, and delete the Session object.

The following sections describe methods common to both Session objects, pubsubsession.p and ptpsession.p.

Creating a Session Procedure

The application creates a session procedure by calling pubsubsession.p or ptpsession.p persistently with the adapterConnection input parameter. At this point, the application has not created the actual JMS session or connected to the SonicMQ Adapter.

The adapterConnection parameter specifies the connection parameters to the SonicMQ Adapter. This allows an application to set different session level attributes before starting the JMS session.

Examples of valid adapterConnection parameters include:

```
"-H host1 -S 5162" /*The Name Server is on host1, listening on UDP port 5162*/
""             /*The Name Server is local, listening on the default UDP port*/
“-URL http://host1:3099/uri?AppService=adapter.progress.jms”
               /*Connecting through HTTP (on a WebClient)*/
“-URL AppServer://host1:5162/uri?AppService=adapter.progress.jms”
               /*Equivalent to "-H host1 -S 5162"*/
```

This is an example of an invalid adapterConnection parameter:

```
 “-URL http://host1:3099/uri”  /*The value adapter’s service name is
missing*/
```

This is the definition of the adapterConnection input parameter to jms/pubsubsession.p and jms/ptpsession.p:

SYNTAX

```
DEFINE INPUT PARAMETER adapterConnection AS CHAR.
```

Setting JMS Connection and Session Attributes

Next, the application specifies connection and session attributes.

The following procedure specifies the service name under which the SonicMQ Adapter is registered with the NameServer. The default is adapter.progress.jms; if the SonicMQ Adapter uses that service name, setAdapterService is unnecessary:

SYNTAX

```
PROCEDURE setAdapterService.
DEFINE INPUT PARAMETER serviceName AS CHAR.
```

The following procedure specifies the JMS broker implementation, SonicMQ. If set on the client side, it overwrites the jmsServerName property set on the SonicMQ Adapter side:

SYNTAX

```
PROCEDURE setJmsServerName.
DEFINE INPUT PARAMETER jmsServerName AS CHAR.
```

The following procedure sets the value of the SonicMQ broker URL. If set on the client, it overwrites the default brokerURL property set on the SonicMQ Adapter side. The creation of a session fails if no value was set on the client or at the SonicMQ Adapter:

SYNTAX

```
PROCEDURE setBrokerURL.
DEFINE INPUT PARAMETER brokerURL AS CHAR.
```

The following procedure specifies the interval in seconds for the SonicMQ Adapter to actively ping the SonicMQ broker so communication failure can be detected promptly. The setPingInterval functionality is a SonicMQ extension. A pingInterval value can also be specified in the srvrStartupParam property of the SonicMQ Adapter as a default for all the clients. No pinging is performed by default. (See the *SonicMQ Programming Guide*). setPingInterval must be called before beginSession is called:

SYNTAX

```
PROCEDURE setPingInterval.
DEFINE INPUT PARAMETER interval AS INT.
```

The following procedure sets the user value for the SonicMQ broker login. If set, it overwrites the default user property set on the SonicMQ Adapter side:

SYNTAX

```
PROCEDURE setUser.
DEFINE INPUT PARAMETER User AS CHAR.
```

The following procedure sets the password value for the SonicMQ broker login. If set, it overwrites the default password property set on the SonicMQ Adapter side:

SYNTAX

```
PROCEDURE setPassword.
DEFINE INPUT PARAMETER password AS CHAR.
```

The following procedure sets the clientID value for the SonicMQ broker connection. If set, it overwrites the default clientID set on the server side. A clientID is required for durable subscriptions:

SYNTAX

```
PROCEDURE setClientID.
DEFINE INPUT PARAMETER clientID AS CHAR.
```

The following procedure makes the session transacted for sending. A session is not transacted by default:

SYNTAX

```
PROCEDURE setTransactedSend.
```

The following procedure makes the session transacted for receiving. A session is not transacted by default:

SYNTAX

```
PROCEDURE setTransactedReceive.
```

Getting JMS Connection and Session Attributes

The following function returns a comma-separated list of connection and provider attributes in this order: JMSVersion, JMSMajorVersion, JMSMinorVersion, JMSProviderName, JMSProviderVersion, JMSProviderMajorVersion, and JMSProviderMinorVersion:

SYNTAX

```
FUNCTION getConnectionMetaData RETURNS CHAR.
```

The following functions return the value set by the preceding set... procedures. Null is returned if the set... procedure was not called; False is returned if setTransacted... was not called:

SYNTAX

```
FUNCTION getJmsServerName RETURNS CHAR.
FUNCTION getAdapterService RETURNS CHAR.
FUNCTION getBrokerURL RETURNS CHAR.
FUNCTION getUser RETURNS CHAR.
FUNCTION getPassword RETURNS CHAR.
FUNCTION getClientID RETURNS CHAR.
FUNCTION getTransactedSend RETURNS LOGICAL.
FUNCTION getTransactedReceive RETURNS LOGICAL.
```

Setting Message Delivery Parameters

Message delivery parameters set on the Session object are used as defaults for all messages sent in that session. The default can be changed by setting the parameters of the publish call, the sendToQueue call, or the requestReply call. These values cannot be changed after beginSession is called.

The following procedure sets the default message priority. The range of priority values is 0 – 9; the default is 4. Setting an UNKNOWN value has no effect:

SYNTAX

```
PROCEDURE setDefaultPriority.
DEFINE INPUT PARAMETER priority AS INT.
```

The following procedure sets the default *time to live*, the value in milliseconds from the time a message is sent to the time the SonicMQ broker can delete the message from the system. A setting of 0 means that the message never expires. The default is JMS broker-dependent; the SonicMQ default value is 0. Any fractional part of the decimal value is truncated. If the value does not fit in a Java long value, Java rules for decimal-to-long conversions are used. Setting an UNKNOWN value has no effect:

SYNTAX

```
PROCEDURE setDefaultTimeToLive.
DEFINE INPUT PARAMETER millis AS DECIMAL.
```

The following procedure sets the message persistency value. The allowed values for message persistency are: PERSISTENT, NON_PERSISTENT, NON_PERSISTENT_ASYNC, and UNKNOWN (?). The default value is PERSISTENT. The evaluation is case insensitive. A call with an UNKNOWN value has no effect. NON_PERSISTENT_ASYNC is a SonicMQ extension of the JMS specification:

SYNTAX

```
PROCEDURE setDefaultPersistency
DEFINE INPUT PARAMETER deliveryMode AS CHAR.
```

Getting Message Delivery Parameters

The following function returns the value specified by setDefaultPriority. It returns 4 if setDefaultPriority was not called:

SYNTAX

```
FUNCTION getDefaultPriority RETURNS INT.
```

The following function returns the value specified by setDefaultTimeToLive. It returns UNKNOWN if setDefaultTimeToLive was not called:

SYNTAX

```
FUNCTION getDefaultTimeToLive RETURNS DECIMAL.
```

The following function returns the value specified by setDefaultPersistency. It returns PERSISTENT if setDefaultPersistency was not called:

SYNTAX

```
FUNCTION getDefaultPersistency RETURNS CHAR.
```

Connecting To the SonicMQ Adapter

To actually connect to the SonicMQ Adapter and start a JMS connection and session, an application calls beginSession. Note that the application does not call beginSession until after setting some (or none) of the previously described attributes. If beginSession returns an error, the Session object is automatically deleted:

SYNTAX

```
PROCEDURE beginSession.
```

NOTE: This procedure executes remotely (sends a message to the SonicMQ Adapter).

Session-level Methods

The following procedure starts receiving messages after creating a new session or after calling stopReceiveMessages. Messages can be sent without calling startReceiveMessages:

SYNTAX

```
PROCEDURE startReceiveMessages.
```

NOTE: This procedure executes remotely (sends a message to the SonicMQ Adapter).

The following procedure causes the SonicMQ Adapter broker to stop receiving messages on behalf of the 4GL client and to stop sending messages already received by the SonicMQ Adapter broker for the client. A subsequent call to startReceiveMessages resumes message reception and delivery.

If this procedure is called in a pubsubsession object and the subscription is not durable, messages published while reception is stopped are not delivered. A single message that was already sent to the client before stopReceiveMessages was called might be received by the client after the stopReceiveMessages call:

SYNTAX

```
PROCEDURE stopReceiveMessages.
```

NOTE: This procedure executes remotely (sends a message to the SonicMQ Adapter).

The following procedure closes a session and its underlying connection and deletes the session procedure:

SYNTAX

```
PROCEDURE deleteSession.
```

NOTE: This procedure executes remotely (sends a message to the SonicMQ Adapter).

The following function returns the AppServer connection ID. This value is typically used to correlate the session to log entries on the server side. This function returns UNKNOWN when called before calling beginSession:

SYNTAX

```
FUNCTION getConnectionID RETURNS CHAR.
```

Load Balancing

Sonic supports the creation of load-balanced clusters. If client-side load balancing is enabled, a connect request can be redirected to another broker within a SonicMQ cluster, provided broker-side load-balancing is enabled.

The following function enables or disables client-side load balancing:

SYNTAX

```
PROCEDURE setLoadBalancing
DEFINE INPUT PARAMETER loadBalancing AS LOGICAL.
```

The following function reports the current status of client-side load balancing, returning TRUE if it is enabled or FALSE if it is disabled:

SYNTAX

```
FUNCTION getLoadBalancing RETURNS LOGICAL.
```

Request/Reply

Request/Reply is a mechanism for the JMSReplyTo message header field to specify the destination where a reply to a message should be sent.The requestReply method sends a message to a destination and designates the messageConsumer parameter for processing replies. Since this call is supported by both the `pubsubsession.p` object and the `ptpsession.p` object, Progress uses the term, destination, which can be used for both topics and queues.

Java–JMS supports a manual approach through the JMSReplyTo field, whereas the 4GL–JMS implementation automates the request/reply sequence:

- Sending a reply by setting the reply OUTPUT parameter of the message handler
- Requesting a reply by calling the requestReply session method with a reply message consumer

The 4GL–JMS implementation uses a temporary destination for the reply. It is an error to set the JMSReplyTo field of the message explicitly if requestReply is used. The reply is received by messageConsumer asynchronously, just like any other message reception. The temporary destination is deleted when the Message Consumer object is deleted:

SYNTAX

```
PROCEDURE requestReply.
DEFINE INPUT PARAMETER destination AS CHAR.
DEFINE INPUT PARAMETER message AS HANDLE.
DEFINE INPUT PARAMETER replySelector AS CHAR.      /*UNKNOWN means
                                                     receiving all replies*/
DEFINE INPUT PARAMETER messageConsumer AS HANDLE.  /*UNKNOWN is illegal*/
DEFINE INPUT PARAMETER priority AS INT.            /*Session default is
                                                     used if UNKNOWN.*/
DEFINE INPUT PARAMETER timeToLive AS DECIMAL.      /*Session default is
                                                     used if UNKNOWN.*/
DEFINE INPUT PARAMETER deliveryMode AS CHAR.       /*Session default is
                                                     used if UNKNOWN.*/
```

NOTE: This procedure executes remotely (sends a message to the SonicMQ Adapter).

Transaction and Recovery Methods

The following procedure sends all messages published (or sent to a queue) up to that point in the current transaction. It is an error to call this method in a Session object that is not transacted for sending:

SYNTAX

```
PROCEDURE commitSend.
```

NOTE: This procedure executes remotely (sends a message to the SonicMQ Adapter).

The following procedure acknowledges all messages received up to that point in the current transaction. It is an error to call this procedure in a Session object that is not transacted for receiving:

SYNTAX

```
PROCEDURE commitReceive.
```

NOTE: This procedure executes remotely (sends a message to the SonicMQ Adapter).

The following procedure discards all messages sent up to that point in the current transaction. It is an error to call this procedure in a Session object that is not transacted for sending:

SYNTAX

```
PROCEDURE rollbackSend.
```

NOTE: This procedure executes remotely (sends a message to the SonicMQ Adapter).

The following procedure starts redelivering the messages received up to that point in the current transaction. It is an error to call this procedure in a Session object that is not transacted for receiving:

SYNTAX

```
PROCEDURE rollbackReceive.
```

NOTE: This procedure executes remotely (sends a message to the SonicMQ Adapter).

The following procedure starts redelivering all unacknowledged messages received up to that point in the current session. It is an error to call this procedure in a Session object that is not transacted for receiving:

SYNTAX

```
PROCEDURE recover.
```

NOTE: This procedure executes remotely (sends a message to the SonicMQ Adapter).

Factory Procedures

Message objects can be created before or after connecting to the SonicMQ Adapter (the beginSession call). This means you can test message manipulation functionality without connecting to the SonicMQ Adapter. Message consumer objects can be called only after beginSession is called.

The following procedure returns a new Message Consumer object in the consumerHandle output parameter:

SYNTAX

```
PROCEDURE createMessageConsumer.
DEFINE INPUT PARAMETER procHandle AS HANDLE.
DEFINE INPUT PARAMETER procName AS CHAR.
DEFINE OUTPUT PARAMETER consumerHandle AS HANDLE.
```

The following procedure creates a new Header Message object in the messageHandle output parameter:

SYNTAX

```
PROCEDURE createHeaderMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.
```

The following procedure creates a new TextMessage object in the messageHandle output parameter:

SYNTAX

```
PROCEDURE createTextMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.
```

The following procedure creates a new MapMessage object in the messageHandle output parameter:

SYNTAX

```
PROCEDURE createMapMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.
```

The following procedure creates a new StreamMessage object in the messageHandle output parameter:

SYNTAX

```
PROCEDURE createStreamMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.
```

The following procedure creates a new BytesMessage object in the messageHandle output parameter:

SYNTAX

```
PROCEDURE createBytesMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.
```

The following procedure creates a new XMLMessage object in the messageHandle output parameter:

SYNTAX

```
PROCEDURE createXMLMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.
```

The following procedure creates a new MultipartMessage object in the messageHandle output parameter:

SYNTAX

```
PROCEDURE createMultipartMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.
```

Fail-over Support

Sonic lets a client specify a list of Sonic brokers to connect to. This makes it easier for the client to connect to a broker when one or more brokers are not available. Sonic also lets the application specify whether to try connecting to the brokers in the list sequentially or randomly.

The following method is called instead of setBrokerURL to specify a list of broker URLs for the client to connect to:

SYNTAX

```
PROCEDURE setConnectionURLs.
DEFINE INPUT PARAMETER brokerList AS CHARACTER.
```

The following method returns a comma-separated list of Sonic broker URLs for the client to try to connect to:

SYNTAX

```
FUNCTION getConnectionURLs RETURNS CHARACTER.
```

The following method lets the application specify whether the brokers in a connection list are tried sequentially or randomly:

SYNTAX

```
PROCEDURE setSequential.
DEFINE INPUT PARAMETER seq AS LOGICAL.
```

The following method reports whether the brokers in a connection list are tried sequentially or randomly:

SYNTAX

```
FUNCTION getSequential RETURNS LOGICAL.
```

Error Handling

Applications should handle asynchronous conditions programmatically by creating a messageConsumer object and passing it to the setErrorHandler procedure in the Session object. Asynchronous conditions are always reported as a TextMessage with several possible CHAR message properties. The CHAR properties that might be included in the message header are: exception, errorCode, linkedException-1, linkedException-2… linkedException-N (where N is a number of additional exceptions linked to the main exception). (See the example in the "Installing an Error Handler To Handle an Asynchronous Error" section in Chapter 14, "Using the SonicMQ Adapter.")

The getPropertyNames message function can be used to get the list of properties in the error message header. If the application does not call setErrorHandler, a default error handler displays the error message and the properties in alert boxes:

SYNTAX

```
PROCEDURE setErrorHandler.
DEFINE INPUT PARAMETER messageConsumer AS HANDLE.
```

NOTE: The application must create the error-handling messageConsumer object and call setErrorHandler after calling beginSession.

The following procedure turns the automatic display of synchronous errors and conditions on and off. If set to true, synchronous errors and conditions are not automatically displayed by the 4GL–JMS implementation. If set to false, synchronous errors and conditions are automatically displayed in alert boxes. The default value is false:

SYNTAX

```
PROCEDURE setNoErrorDisplay.
DEFINE INPUT PARAMETER noDisplay AS LOGICAL.
```

NOTE: Messages inherit the display/noDisplay property from the session that created them. However, after a message is created, it is independent from the session; setNoErrorDisplay must be called in the message itself to change the property.

Processing Messages

The waitForMessages call waits and processes events as long as the user-defined function UDFName (in procH) returns true and there is no period of more than timeOut seconds in which no messages are received. The user-defined function is evaluated each time after a message is handled:

SYNTAX

```
PROCEDURE waitForMessages:
DEFINE INPUT PARAMETER UDFName AS CHAR NO-UNDO.
DEFINE INPUT PARAMETER procH AS HANDLE NO-UNDO.
DEFINE INPUT PARAMETER timeOut AS INT NO-UNDO.
```

Flow Control

When the SonicMQ Adapter sends a message to a queue that is full or to a topic whose queue is full, an error is raised.

Also, the SonicMQ Adapter lets the client set the prefetch count and the prefetch threshold, which are used when the client retrieves messages from a queue. The prefetch count sets how many messages the Sonic client can retrieve in a single operation from a queue that contains multiple messages. The prefetch threshold determines when the Sonic client goes back to the broker for more messages. For example, a threshold of one means that the client does not go back to the broker until the last message has been delivered.

The following method sets the prefetch count:

SYNTAX

```
PROCEDURE setPrefetchCount.
DEFINE INPUT PARAMETER count AS INTEGER.
```

The following method sets the prefetch threshold:

SYNTAX

```
PROCEDURE setPrefetchThreshold.
DEFINE INPUT PARAMETER threshold AS INTEGER.
```

Comparison Of pubsubsession.p With ptpsession.p

The persistent procedures, `pubsubsession.p` and `ptpsession.p`, have the same methods except:

- **Message sending** — Pub/Sub uses the publish method and PTP uses the sendToQueue method.
- **Message receiving** — Pub/Sub uses the subscribe method and PTP uses the receiveFromQueue method.
- **Queue browsing** — PTP uses the browseQueue method to browse messages on a queue; there is no equivalent in the Pub/Sub model.

NOTES:

- The requestReply method can be used with both a `pubsubsession.p` session and a `ptpsession.p` session.
- The cancelDurableSubscription method can be used only with the `pubsubsession.p` session, since the concept of a durable subscription does not exist in the PTP domain.

pubsubsession.p

Most methods are common to both `pubsubsession.p` and `ptpsession.p`. This section describes only those methods unique to `pubsubsession.p`.

An application creates the session procedure for Pub/Sub messaging by calling `pubsubsession.p` persistently with the following input parameter:

SYNTAX

```
DEFINE INPUT adapterConnection AS CHAR.
```

For more information about adapterConnection, see the "Creating a Session Procedure" section in this chapter.

Publishing To a Topic

An application uses the publish procedure to publish messages to a topic. The publish procedure takes five input parameters:

- The name of a topic
- A Message object
- Message priority: 0 – 9 (optional)
- TimeToLive in milliseconds (optional)
- Delivery mode (optional)

The publish procedure publishes a message to topicName. If the publication is in reply to a received message, topicName can be the replyTo field obtained from the original message:

SYNTAX

```
PROCEDURE publish.
DEFINE INPUT PARAMETER topicName AS CHAR.
DEFINE INPUT PARAMETER message AS HANDLE.
DEFINE INPUT PARAMETER priority AS INT.             /*Session default is
                                                      used if UNKNOWN.*/
DEFINE INPUT PARAMETER timeToLive AS DECIMAL.       /*Session default is
                                                      used if UNKNOWN.*/
DEFINE INPUT PARAMETER deliveryMode AS CHAR.        /*Session default is
                                                      used if UNKNOWN.*/
```

NOTE: This procedure executes remotely (sends a message to the SonicMQ Adapter).

Subscribing To a Topic

An application uses the subscribe procedure to subscribe to a topic. The subscribe procedure takes five input parameters:

- The name of a topic
- A name of a durable subscription (optional)
- A message selector (optional)
- Specifying not to receive its own messages (optional)
- A Message Consumer object

The subscribe procedure subscribes to topicName. The messages are handled asynchronously by the messageConsumer object. A subscriptionName parameter with a value other than UNKNOWN specifies a durable subscription. Durable subscriptions require the JMS client to have a clientID identifier. The client must call setClientID in the `pubsubsession.p` object (or set the default clientID on the server side) if a durable subscription is desired. If the subscriptionName value is UNKNOWN or an empty string, the subscription is not durable. The default of noLocalPublications is false; the session, by default, gets its own publications:

SYNTAX

```
PROCEDURE subscribe.
DEFINE INPUT PARAMETER topicName AS CHAR.
DEFINE INPUT PARAMETER subscriptionName AS CHAR.
DEFINE INPUT PARAMETER messageSelector AS CHAR.
DEFINE INPUT PARAMETER noLocalPublications AS LOGICAL.
DEFINE INPUT PARAMETER messageConsumer AS HANDLE.
```

NOTE: This procedure executes remotely (sends a message to the SonicMQ Adapter).

The following procedure cancels a durable subscription. It is an error to call this procedure if there is an active message consumer for that subscription; call deleteConsumer first to delete the message consumer:

SYNTAX

```
PROCEDURE cancelDurableSubscription.
DEFINE INPUT PARAMETER subscriptionName AS CHAR.
```

NOTE: This procedure executes remotely (sends a message to the SonicMQ Adapter).

ptpsession.p

Most Session methods are common to both `pubsubsession.p` and `ptpsession.p`. This section describes only those methods unique to `ptpsession.p`.

Applications use `ptpsession.p`. for PTP messaging. The session procedure for PTP messaging is created by calling `ptpsession.p` persistently with the following input parameter:

SYNTAX

```
DEFINE INPUT adapterConnection AS CHAR.
```

For more information about adapterConnection, see the "Creating a Session Procedure" section in this chapter.

Sending Messages To a Queue

Applications use the sendToQueue procedure to send messages to a queue. The sendToQueue procedure takes five input parameters:

- The name of a queue
- A Message object
- Message priority: 0–9 (optional)
- TimeToLive in milliseconds (optional)
- Delivery mode (optional)

This procedure sends a message to queueName. If the sending is in reply to a received message, queueName can be the replyTo field obtained from the original message:

SYNTAX

```
PROCEDURE sendToQueue.
DEFINE INPUT PARAMETER queueName AS CHAR.
DEFINE INPUT PARAMETER message AS HANDLE.
DEFINE INPUT PARAMETER priority AS INT.              /*Session default is
                                                       used if UNKNOWN.*/
DEFINE INPUT PARAMETER timeToLive AS DECIMAL.        /*Session default is
                                                       used if UNKNOWN.*/
DEFINE INPUT PARAMETER deliveryMode AS CHAR.         /*Session default is
                                                       used if UNKNOWN.*/
```

NOTE: This procedure executes remotely (sends a message to the SonicMQ Adapter).

Receiving Messages From a Queue

Applications use receiveFromQueue to receive messages from a queue. The receiveFromQueue procedure takes three input parameters:

- The name of a queue
- A message selector (optional)
- A Message Consumer object

This procedure receives messages from queueName. The messages are handled asynchronously by the messageConsumer procedure:

SYNTAX

```
PROCEDURE receiveFromQueue.
DEFINE INPUT PARAMETER queueName AS CHAR.
DEFINE INPUT PARAMETER messageSelector AS CHAR.   *UNKNOWN means
                                                   receiving all messages.*/
DEFINE INPUT PARAMETER messageConsumer AS HANDLE.
```

NOTE: This procedure executes remotely (sends a message to the SonicMQ Adapter).

Browsing Messages On a Queue

Applications use browseQueue to view messages in a queue without consuming them. This procedure receives (for browsing) all messages currently in the queue in the messageConsumer object:

SYNTAX

```
PROCEDURE browseQueue.
DEFINE INPUT PARAMETER queueName AS CHAR.
DEFINE INPUT PARAMETER messageConsumer AS HANDLE.
```

NOTES:

- Browsed messages are not removed from the queue or acknowledged and are not subject to the transactional context of the session. (For more information on queue browsing, see the *Java Message Service* specification and the *SonicMQ Programming Guide*.)
- The session does not have to run startReceiveMessages to browse messages on a queue.
- This procedure executes remotely (sends a message to the SonicMQ Adapter).

13.3.2 Message Consumer Objects

The 4GL application uses a Message Consumer to receive messages from a destination or to receive asynchronous error messages.

A single Message Consumer object can be passed:

- Once to a subscribe call
- Once to a receiveFromQueue call
- Once to a browseQueue call
- Many times to requestReply methods to serve as a reply handler
- Once to setErrorHandler to consume asynchronous error messages

The life cycle of a Message Consumer object includes these general steps:

1. An application implements a procedure to handle the messages.
2. The application creates the message consumer, specifying the message-handling procedure.
3. The application uses the Message Consumer object to do one of the following: subscribe (Pub/Sub) or receive (PTP) messages from the queue, set it an error handle and receive error messages asynchronously from SonicMQ through the SonicMQ Adapter, or receive replies in a request/reply cycle.
4. After using the Message Consumer object, the application can activate it by getting into a WAIT FOR state (or any IO blocking state where the application processes events).
5. When the Message Consumer finishes processing all of the messages of interest, the application calls deleteConsumer to release the resources in the 4GL application and the SonicMQ Adapter and SonicMQ broker.

An application creates a messageConsumer object by calling createMessageConsumer in a Session object. The application must pass to createMessageConsumer, the name of an internal procedure (procName) for handling messages and a handle to a procedure that contains procName (procHandle). The new Message Consumer object is returned in consumerHandle:

SYNTAX

```
PROCEDURE createMessageConsumer.
DEFINE INPUT PARAMETER procHandle AS HANDLE.
DEFINE INPUT PARAMETER procName AS CHAR.
DEFINE OUTPUT PARAMETER consumerHandle AS HANDLE.
```

Setting Context

The 4GL application can use setApplicationContext to pass context to the message handler. The handler parameter is typically a handle to a persistent procedure implemented by the application. When the message handler is called, it gets that handler and uses it, for example, to deposit error information in the application's context by calling a specific handler's internal procedure:

SYNTAX

```
PROCEDURE setApplicationContext.
DEFINE INPUT PARAMETER handler AS HANDLE.
```

Setting Reply Properties

The following procedures set reply properties when the message consumer is passed to requestReply.

The following procedure sets the priority of the reply messages; the range of values is 0–9; the default is 4. The setReplyPriority procedure can be called only once:

SYNTAX

```
PROCEDURE setReplyPriority.
DEFINE INPUT PARAMETER priority AS INT.
```

The following procedure sets the time to live value. The default is JMS system-dependent; the SonicMQ default value is 0. The replyTimeToLive values can be set only once. The fractional part of the decimal value is truncated. If the value does not fit in a Java long value, Java rules for decimal-to-long conversion apply:

SYNTAX

```
PROCEDURE setReplyTimeToLive.
DEFINE INPUT PARAMETER millis AS DECIMAL.
```

The following procedure sets the value for message persistency. The allowed values are: PERSISTENT, NON_PERSISTENT, NON_PERSISTENT_ASYNC, and UNKNOWN; the default value is PERSISTENT. The evaluation is case insensitive. A call with an UNKNOWN value has no effect. The replyPersistency value can be set only once:

SYNTAX

```
PROCEDURE setReplyPersistency.
DEFINE INPUT PARAMETER deliveryMode AS CHAR.
```

NOTE: NON_PERSISTENT_ASYNC is a SonicMQ extension.

The following procedure specifies whether all reply messages are to be automatically deleted. If set to true, then all reply messages returned through the message handler's OUPUT parameter are automatically deleted after being sent. The default value is false: (See the "Message Handler" section in this chapter.)

SYNTAX

```
PROCEDURE setReplyAutoDelete.
DEFINE INPUT PARAMETER val AS LOGICAL.
```

Getting Message Handler Properties

The following functions return the properties of the message handler and the application context:

SYNTAX

```
FUNCTION getProcHandle RETURNS HANDLE.
FUNCTION getProcName RETURNS CHAR.
FUNCTION getApplicationContext RETURNS HANDLE.
```

The following function returns the name of the destination that messages arrive from when the message consumer was passed to subscribe or to receiveFromQueue:

SYNTAX

```
FUNCTION getDestinationName RETURNS CHAR.
```

One of the following functions returns true when called from a message handler. That allows the application to determine whether it is handling an error message, a subscription (or queue) message, a reply message, or a queue browsing message. The value returned from the function when the application is not currently in any of these message handling states is not determined:

SYNTAX

```
FUNCTION inErrorHandling RETURNS LOGICAL.
FUNCTION inMessageHandling RETURNS LOGICAL.
FUNCTION inReplyHandling RETURNS LOGICAL.
FUNCTION inQueueBrowsing RETURNS LOGICAL.
```

The following function returns a handle to the session:

SYNTAX

```
FUNCTION getSession RETURNS HANDLE.
```

The following function returns the setReplyPriority value (4 if setReplyPriority was not called):

SYNTAX

```
FUNCTION getReplyPriority RETURNS INT.
```

The following function returns the setReplyTimeToLive value (UNKNOWN if setReplyTimeToLive was not called):

SYNTAX

```
FUNCTION getReplyTimeToLive RETURNS DECIMAL.
```

The following function returns the setReplyPersistency value (PERSISTENT if setReplyPersistency was not called):

SYNTAX

```
FUNCTION getReplyPersistency RETURNS CHAR.
```

The following function returns the value set by setReplyAutoDelete:

SYNTAX

```
Function getReplyAutoDelete RETURNS LOGICAL.
```

Terminating the Message Consumer

The following procedure ends the life of the Message Consumer object. It cancels the subscription (in the Pub/Sub domain) or the association with a queue (in the PTP domain) and deletes the messageConsumer Object:

SYNTAX

```
PROCEDURE deleteConsumer.
```

NOTE: This procedure executes remotely (sends a message to the SonicMQ Adapter).

Acknowledge and Forward

Imagine the following scenario:

1. A client retrieves a message from a broker's queue.
2. The broker wants to be notified when the message reaches its ultimate destination.
3. The ultimate destination is a remote queue.
4. The client sends the message on its way.

To acknowledge receipt of a message whose ultimate destination is a remote queue, you might enclose the message and acknowledgment in a single transaction. But this introduces the overhead and complexity of transaction processing. SonicMQ provides a cleaner solution, embodied in the following steps:

1 ♦ Run the setSingleMessageAcknowledgement method to set the session to SINGLE_MESSAGE_ACKNOWLEDGE.

2 ♦ Run the acknowledgeAndForward method within the message event handler, specifying a destination queue name, the original message handle, and optional message-delivery properties (priority, time-to-live, and persistency). If the method is successful, the message is forwarded and acknowledged in a single, atomic operation.

The singleMessageAcknowledgement method sets the acknowledgement setting of a client session. Its syntax is:

SYNTAX

```
PROCEDURE setSingleMessageAcknowledgement.
DEFINE INPUT PARAMETER ackMethod as LOGICAL.
```

The getSingleMessageAcknowledgement method returns the acknowledgement setting of a client session. Its syntax is:

SYNTAX

```
FUNCTION getSingleMessageAcknowledegment RETURNS LOGICAL.
```

The acknowledgeAndForward method, to be run within a message event handler, causes a message to be forwarded and acknowledged in a single, atomic operation. Its syntax is:

SYNTAX

```
PROCEDURE acknowledgeAndForward.
DEFINE INPUT PARAMETER destinationName AS CHARACTER.
DEFINE INPUT PARAMETER messageH        AS HANDLE.
DEFINE INPUT PARAMETER priority        AS INTEGER.
DEFINE INPUT PARAMETER timeToLive      AS DECIMAL.
DEFINE INPUT PARAMETER persistency     AS CHARACTER.
```

Stopping Acknowledgement

The following procedure instructs the 4GL–JMS implementation not to acknowledge this message. This call should be made, for example, if the 4GL application fails to use the data in a message and needs to receive the message again. This call is an error if the session is transacted for receiving. If the messageConsumer object is used to handle error messages or for queue browsing, this call has no effect:

SYNTAX

```
PROCEDURE setNoAcknowledge.
```

The following function returns true if setNoAcknowledge was called:

SYNTAX

```
FUNCTION getNoAcknowledge RETURNS LOGICAL.
```

Specifying Message Reuse

The following procedure instructs the Message Consumer object not to create a new message for each received message. Calling setReuseMessage improves performance: if the procedure is not called, the Message Consumer object creates a new message for each received message. A message that is being reused should not be deleted before the session is deleted:

SYNTAX

```
PROCEDURE setReuseMessage.
```

The following function returns true if setReuseMessage was called; if not, it returns false:

SYNTAX

```
FUNCTION getReuseMessage RETURNS LOGICAL.
```

13.3.3 Message Objects

The following sections describe the functions and procedures of the Message Objects, including message headers, message properties, and the various message types.

Message Header Interface

The message header provides envelope information about a message. The message header interface is supported by all message types and all message types have the same header information. Header information is the same for both PTP and Pub/Sub messaging.

The message header is not created directly by the application. When any type of message is created, its header procedure is automatically created. The message procedure delegates header method calls to its header procedure.

All the fields of the header can be seen by either the sender or receiver:

- A receiver can read header information set by the sender or by the JMS system when it receives the message.
- A sender can see header information set by the application immediately after setting the header information and see information set by the JMS system (for example, the messageID) after sending a message.

Setting Message Header Information

Header information is:

- Set by the JMS System. For example, the messageID, timestamp, and reDelivered.
- Calculated by the JMS system but derived from information set by the application. For example, if the application sets the time to live, the JMS system calculates the expiration time.
- Set explicitly by the application. For example, the replyTo destination and correlationID.

The following procedure sets a destination for replies. Note that the destination can be a name of a queue even if the message is sent by a Pub/Sub session and the destination can be the name of the topic even if the message is sent by a PTP session. However, in that case, setReplyToDestinationType must be called to set the correct destination type:

SYNTAX

```
PROCEDURE setJMSReplyTo
DEFINE INPUT PARAMETER destination AS CHAR.
```

The following procedure sets the type of the destination specified by setJMSReplyTo; the type can be “queue” or “topic” If setReplyToDestinationType is not called, a default type is automatically set when the message is sent, according to the type of the session:

SYNTAX

```
PROCEDURE setReplyToDestinationType
DEFINE INPUT PARAMETER type AS CHAR.
```

The following procedure sets the correlationID. This value is application-defined; typically it is set to the ID of the message replied to:

SYNTAX

```
PROCEDURE setJMSCorrelationID
DEFINE INPUT PARAMETER correlationID AS CHAR.
```

The following procedure sets the bytes correlationID; the bytesCorrelation ID usage is proprietary (JMS provider-dependent). (A *JMS provider* is a messaging system that implements the JMS interface.) When accessing SonicMQ, the bytesCorrelationID field can be used for storing application-defined values:

SYNTAX

```
PROCEDURE setJMSCorrelationIDAsBytes
DEFINE INPUT PARAMETER bytesCorrelationID AS RAW.
```

The following procedure sets the type name; the type name is proprietary (JMS provider-dependent). When accessing SonicMQ, the JMSType field can be used for storing application-defined values:

SYNTAX

```
PROCEDURE setJMSType
DEFINE INPUT PARAMETER typeName AS CHAR.
```

Getting Message Header Information

The following functions return message header information.

The following function returns the SonicMQ Adapter message type: TextMessage, MapMessage, StreamMessage, BytesMessage, HeaderMessage, or XMLMessage:

SYNTAX

```
FUNCTION getMessageType RETURNS CHAR.
```

The following function returns the message ID, a unique ID that the JMS server assigns to each message:

SYNTAX

```
FUNCTION getJMSMessageID RETURNS CHAR.
```

The following function returns the message sending time, which is the difference in milliseconds between the message creation time and midnight, January 1, 1970 UTC:

SYNTAX

```
FUNCTION getJMSTimestamp RETURNS DECIMAL.
```

The following function returns the name of the destination this message was sent to. The value is valid after the message was sent (at the sender side) and in the received message (at the receiver side):

SYNTAX

```
FUNCTION getJMSDestination RETURNS CHAR.
```

The following function returns true (at the receiver side) if this is not the first delivery of this message. A second delivery can take place if the first delivery is not acknowledged by the receiver or the transaction was rolled back, in a transacted session:

SYNTAX

```
FUNCTION getJMSRedelivered RETURNS LOGICAL.
```

The following function returns the correlationID. This value is application-defined, typically the ID of the message replied to:

SYNTAX

```
FUNCTION getJMSCorrelationID RETURNS CHAR.
```

The following function returns a proprietary (JMS provider-dependent) correlation ID. When accessing SonicMQ, the bytesCorrelationID field can be used for storing application-defined values:

SYNTAX

```
FUNCTION getJMSCorrelationIDAsBytes RETURNS RAW.
```

The following function returns the delivery mode. This value is PERSISTENT, NON_PERSISTENT, or DISCARDABLE. The message receiver never gets the NON_PERSISTENT_ASYNC value. A message sent using NON_PERSISTENT_ASYNC is received with the standard NON_PERSISTENT value:

SYNTAX

```
FUNCTION getJMSDeliveryMode RETURNS CHAR.
```

The following function returns the reply destination. The destination can be the name of a queue, even if the message is received from a Pub/Sub session, and the destination can be the name of a topic even if the message is received from a PTP session. The replyToDestinationType function must be called if both a queue destination and a topic destination might be stored in the received message:

SYNTAX

```
FUNCTION getJMSReplyTo RETURNS CHAR.
```

The following function returns “queue,” “topic,” or UNKNOWN. The UNKNOWN value is returned if the message was just created, not sent yet, and setReplyToDestinationType was not called. Call getReplyToDestinationType when the domain of the ReplyTo field is not known:

SYNTAX

```
FUNCTION getReplyToDestinationType RETURNS CHAR.
```

The following function returns a proprietary (JMS provider-dependent) type name. When accessing SonicMQ, the JMSType field can be used for storing application-defined values:

SYNTAX

```
FUNCTION getJMSType RETURNS CHAR.
```

The following function returns the expiration time (GMT):

SYNTAX

```
FUNCTION getJMSExpiration RETURNS DECIMAL.
```

The following function returns priority values in the range of 0–9:

SYNTAX

```
FUNCTION getJMSPriority RETURNS int.
```

The following function returns true if the JMSReplyTo header was set:

SYNTAX

```
FUNCTION hasReplyTo RETURNS LOGICAL.
```

Message Properties

Message properties can add more envelope information about a message. There is a fixed number of header fields, but properties are flexible. An application can add any number of properties—property name and value pairs.

The following function returns the message property's data type. UNKNOWN is returned if the property was not set in the message:

SYNTAX

```
FUNCTION getPropertyType RETURNS CHAR (propertyName AS CHAR).
```

The following function returns a comma-separated list of the properties of the message:

SYNTAX

```
FUNCTION getPropertyNames RETURNS CHAR.
```

Setting Message Properties

The following procedures set message properties.

NOTE: The server returns a NumberFormatException message for value overflows (for example, if setByteProperty("prop1", 1000) is called).

The following procedure sets a boolean message property. An UNKNOWN value is considered a false value:

SYNTAX

```
PROCEDURE setBooleanProperty.
DEFINE INPUT PARAMETER propertyName AS CHAR.
DEFINE INPUT PARAMETER propertyValue AS LOGICAL.
```

The following procedure sets a bytes message property; the values range from –128 to 127:

SYNTAX

```
PROCEDURE setByteProperty.
DEFINE INPUT PARAMETER propertyName AS CHAR.
DEFINE INPUT PARAMETER propertyValue AS INT.
```

The following procedure sets a short message property:

SYNTAX

```
PROCEDURE setShortProperty.
DEFINE INPUT PARAMETER propertyName AS CHAR.
DEFINE INPUT PARAMETER propertyValue AS INT.
```

The following procedure sets an int message property:

SYNTAX

```
PROCEDURE setIntProperty.
DEFINE INPUT PARAMETER propertyName AS CHAR.
DEFINE INPUT PARAMETER propertyValue AS INT.
```

The following procedure sets a long message property; any fractional part of the DECIMAL value is truncated:

SYNTAX

```
PROCEDURE setLongProperty.
DEFINE INPUT PARAMETER propertyName AS CHAR.
DEFINE INPUT PARAMETER propertyValue AS DECIMAL.
```

The following procedure sets a float message property:

SYNTAX

```
PROCEDURE setFloatProperty.
DEFINE INPUT PARAMETER propertyName AS CHAR.
DEFINE INPUT PARAMETER propertyValue AS DECIMAL.
```

The following procedure sets a double message property:

SYNTAX

```
PROCEDURE setDoubleProperty.
DEFINE INPUT PARAMETER propertyName AS CHAR.
DEFINE INPUT PARAMETER propertyValue AS DECIMAL.
```

The following procedure sets a String message property:

SYNTAX

```
PROCEDURE setStringProperty.
DEFINE INPUT PARAMETER propertyName AS CHAR.
DEFINE INPUT PARAMETER propertyValue AS CHAR.
```

The following procedure clears the body of a message and keeps header and property values unchanged. This procedure transfers a StreamMessage, TextMessage, XMLMessage, and a BytesMessage to write-only mode:

SYNTAX

```
PROCEDURE clearBody.
```

The following procedure clears the properties of the message, keeping header and body values unchanged:

SYNTAX

```
PROCEDURE clearProperties.
```

Getting Message Properties

The following function returns a boolean message property:

SYNTAX

```
FUNCTION getLogicalProperty RETURNS LOGICAL (propertyName AS CHAR).
```

The following function returns int, short, and byte message properties:

SYNTAX

```
FUNCTION getIntProperty RETURNS INT (propertyName AS CHAR).
```

The following function returns any numeric message property:

SYNTAX

```
FUNCTION getDecimalProperty RETURNS DECIMAL (propertyName AS CHAR).
```

The following function returns a message property of any data type:

SYNTAX

```
FUNCTION getCharProperty RETURNS CHAR (propertyName AS CHAR).
```

Deleting Messages

The following procedure deletes a message:

SYNTAX

```
PROCEDURE deleteMessage.
```

Maximum Number Of Messages

The default maximum number of JMS messages in a 4GL session is 50 (the total number of messages created by the application plus messages received from JMS). To change the default to *new–val*, the following definition must be included in the main procedure of the 4GL application:

SYNTAX

```
DEFINE NEW GLOBAL SHARED VAR JMS-MAXIMUM-MESSAGES AS INT INIT new-val.
```

Error Display

If setNoErrorDisplay is set to true, messages from errors that occur when methods in a message are called are not automatically displayed. The default value is false. Messages inherit the display/noDisplay property from the session that created them. However, after the message is created, it is independent from the session, and setNoErrorDisplay must be called in the message itself to change the display/noDisplay property:

SYNTAX

```
PROCEDURE setNoErrorDisplay.
DEFINE INPUT PARAMETER noDisplay AS LOGICAL.
```

Message Handler

The message handler is written by an application and must be registered with the Message Consumer object. When a message is received, the message handler is called automatically so the application can process the message.

The messageHandler parameters are:

- **message** — The message
- **messageConsumer** — The Message Consumer object that contains this message handler. The application can use the Message Consumer object to get context information about the message (for example, the session handle to the session that received that message) and the context (for example, the session handler).
- **reply** — The application can reply to the message automatically without having to extract the reply to fields. The application can set the reply parameter with a reply message, and it is automatically sent to the JMSReplyTo destination of the message.

  If the setReplyAutoDelete(true) message consumer procedure is called, the reply message is automatically deleted after being sent.

The 4GL programmer implements procedures with the following signature for handling incoming messages (JMS messages and error messages):

SYNTAX

```
PROCEDURE messageHandler.
DEFINE INPUT PARAMETER message AS HANDLE.
DEFINE INPUT PARAMETER messageConsumer AS HANDLE.
DEFINE OUTPUT PARAMETER reply AS HANDLE.
```

TextMessage

An application creates a TextMessage by calling createTextMessage in a Session object with the following output parameter:

SYNTAX

```
PROCEDURE createTextMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.
```

The following function returns the total number of characters in the message:

SYNTAX

```
FUNCTION getCharCount RETURNS INT.
```

The following function returns true if the last text segment was retrieved:

SYNTAX

```
FUNCTION endOfStream RETURNS LOGICAL.
```

When a TextMessage is created, it is in a write-only mode. Calling reset changes the mode to read-only and places the cursor before the first text segment. Sending the message causes an implicit reset and the message becomes read-only. The message arrives at the receiver in a reset state:

SYNTAX

```
PROCEDURE reset.
```

Setting Text

The following procedure clears the message body and sets a new text value. The call can be made when the message is in write-only or read-only mode. After the call, the message is in write-only mode and additional appendText calls can be made to append more text:

SYNTAX

```
PROCEDURE setText.
DEFINE INPUT PARAMETER textValue AS CHAR.
```

The following procedure can be called in write-only mode to append text to the message in several calls to overcome the Progress 32K limit on the number of characters:

SYNTAX

```
PROCEDURE appendText.
DEFINE INPUT PARAMETER textValue AS CHAR.
```

Getting Text

The following function returns all the text in the TextMessage and then implicitly calls reset. A run-time error occurs if the size of the message is too large to be handled by the 4GL interpreter:

SYNTAX

```
FUNCTION getText RETURNS CHAR.
```

The following function can be called in read-only mode to return the next text segment when handling large messages:

SYNTAX

```
FUNCTION getTextSegment RETURNS CHAR.
```

Header Messages

A *HeaderMessage* is a message type that is the 4GL–JMS equivalent of Java's `javax.jms.Message`, a header-only message. It sends information through message properties without a message body.

An application creates a HeaderMessage by calling createHeaderMessage in a Session object with the following output parameter:

SYNTAX

```
PROCEDURE createHeaderMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.
```

StreamMessage

A StreamMessage allows applications to send and receive an unspecified number of items; each item is a Java data type. All basic Java data types are supported. When receiving any arbitrary Java data type, an application uses methods to read and specify a Progress data type. When writing a message from the Progress 4GL, an application uses methods to send any of those Java data types and to specify the data.

To create a StreamMessage, an application calls createStreamMessage in a session object with the following output parameter:

SYNTAX

```
PROCEDURE createStreamMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.
```

When a StreamMessage is created, it is in a write-only mode. The following procedure changes the mode to read-only. Sending the message causes an implicit reset. The message arrives at the receiver in a reset mode:

SYNTAX

```
PROCEDURE reset.
```

The following function returns true if the last item of the stream was retrieved:

SYNTAX

```
FUNCTION endOfStream RETURNS LOGICAL.
```

Writing Data To a StreamMessage

The following procedures write data to the body of a StreamMessage.

NOTE: The server returns a NumberFormatException message for a value overflow, for example, if writeByte(1000) is called.

The following procedure writes boolean data to the body of a StreamMessage. An UNKNOWN value is considered false:

SYNTAX

```
PROCEDURE writeBoolean.
DEFINE INPUT PARAMETER value AS LOGICAL.
```

The following procedure writes bytes data to the body of a StreamMessage; byte values are –128 to 127:

SYNTAX

```
PROCEDURE writeByte.
DEFINE INPUT PARAMETER value AS INT.
```

The following procedure writes short data to the body of a StreamMessage:

SYNTAX

```
PROCEDURE writeShort.
DEFINE INPUT PARAMETER value AS INT.
```

The following procedure writes character data to the body of a StreamMessage; the number of characters in the CHAR value must be one:

SYNTAX

```
PROCEDURE writeChar.
DEFINE INPUT PARAMETER value AS CHAR.
```

The following procedure writes integer data to the body of a StreamMessage:

SYNTAX

```
PROCEDURE writeInt.
DEFINE INPUT PARAMETER value AS INT.
```

The following procedure writes long data to the body of a StreamMessage. The fractional part of the DECIMAL value is truncated:

SYNTAX

```
PROCEDURE writeLong.
DEFINE INPUT PARAMETER value AS DECIMAL.
```

The following procedure writes float data to the body of a StreamMessage:

SYNTAX

```
PROCEDURE writeFloat.
DEFINE INPUT PARAMETER value AS DECIMAL.
```

The following procedure writes double data to the body of a StreamMessage:

SYNTAX

```
PROCEDURE writeDouble.
DEFINE INPUT PARAMETER value AS DECIMAL.
```

The following procedure writes String data to the body of a StreamMessage:

SYNTAX

```
PROCEDURE writeString.
DEFINE INPUT PARAMETER value AS CHAR.
```

The following procedure writes bytes data to the body of a StreamMessage:

SYNTAX

```
PROCEDURE writeBytesFromRaw.
DEFINE INPUT PARAMETER value AS RAW.
```

Reading Data From a StreamMessage

The following functions read data from the body of a StreamMessage.

The following function moves the cursor to the next data item and returns its data type, one of the following values: UNKNOWN, boolean, byte, short, char, int, long, float, double, string, or bytes. UNKNOWN is returned when the value of the item is NULL. When the message is received or after reset is called, the cursor is set before the first data item. It is an error to try to move the cursor beyond the last item:

SYNTAX

```
FUNCTION moveToNext RETURNS CHAR.
```

The following function returns boolean data from the body of a StreamMessage:

SYNTAX

```
FUNCTION readLogical RETURNS LOGICAL.
```

The following function returns int, short, or bytes data from the body of a StreamMessage:

SYNTAX

```
FUNCTION readInt RETURNS INT.
```

The following function returns any numeric data from the body of a StreamMessage:

SYNTAX

```
FUNCTION readDecimal RETURNS DECIMAL.
```

The following function returns any data except bytes data from the body of a StreamMessage:

SYNTAX

```
FUNCTION readChar RETURNS CHAR.
```

The following function returns bytes data from the body of a StreamMessage:

SYNTAX

```
FUNCTION readBytesToRaw RETURNS RAW.
```

MapMessage

A *MapMessage* contains multiple pairs of item names and item values. When setting the data, the application specifies the Java data type. When getting the data, the application specifies the 4GL data type to convert into. For more information on mapping data types, see the "Data Storage and Extraction Methods" section in this chapter.

An application creates a MapMessage by calling createMapMessage in a Session object with the following output parameter:

SYNTAX

```
PROCEDURE createMapMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.
```

The following function returns a comma-separated list of the item names in a MapMessage:

SYNTAX

```
FUNCTION getMapNames RETURNS CHAR.
```

The following function returns the data type of an item in a MapMessage. UNKNOWN is returned if the item does not exist:

SYNTAX

```
FUNCTION getItemType RETURNS CHAR (itemName AS CHAR).
```

Setting Items In a MapMessage

A MapMessage supports setting multiple named item pairs: {item-name, item-value}. The following procedures convert data from 4GL to JMS data types.

NOTE: The server returns a NumberFormatException message for a value overflow, for example, if setByte(“item1”, 1000) is called.

The following procedure sets boolean data in a MapMessage. An UNKNOWN value is considered false:

SYNTAX

```
PROCEDURE setBoolean.
DEFINE INPUT PARAMETER itemName AS CHAR.
DEFINE INPUT PARAMETER value AS LOGICAL.
```

The following procedure sets bytes data in a MapMessage; byte values are –128 to 127:

SYNTAX

```
PROCEDURE setByte.
DEFINE INPUT PARAMETER itemName AS CHAR.
DEFINE INPUT PARAMETER value AS INT.
```

The following procedure sets short data in a MapMessage:

SYNTAX

```
PROCEDURE setShort.
DEFINE INPUT PARAMETER itemName AS CHAR.
DEFINE INPUT PARAMETER value AS INT.
```

The following procedure sets character data in a MapMessage; the number of characters in the CHAR value must be one:

SYNTAX

```
PROCEDURE setChar.
DEFINE INPUT PARAMETER itemName AS CHAR.
DEFINE INPUT PARAMETER value AS CHAR.
```

The following procedure sets integer data in a MapMessage:

SYNTAX

```
PROCEDURE setInt.
DEFINE INPUT PARAMETER itemName AS CHAR.
DEFINE INPUT PARAMETER value AS INT.
```

The following procedure sets long data in a MapMessage; any fractional part of the DECIMAL value is truncated:

SYNTAX

```
PROCEDURE setLong.
DEFINE INPUT PARAMETER itemName AS CHAR.
DEFINE INPUT PARAMETER value AS DECIMAL.
```

The following procedure sets float data in a MapMessage:

SYNTAX

```
PROCEDURE setFloat.
DEFINE INPUT PARAMETER itemName AS CHAR.
DEFINE INPUT PARAMETER value AS DECIMAL.
```

The following procedure sets double data in a MapMessage:

SYNTAX

```
PROCEDURE setDouble.
DEFINE INPUT PARAMETER itemName AS CHAR.
DEFINE INPUT PARAMETER value AS DECIMAL.
```

The following procedure sets String data in a MapMessage:

SYNTAX

```
PROCEDURE setString.
DEFINE INPUT PARAMETER itemName AS CHAR.
DEFINE INPUT PARAMETER value AS CHAR.
```

The following procedure sets bytes data in a MapMessage:

SYNTAX

```
PROCEDURE setBytesFromRaw.
DEFINE INPUT PARAMETER itemName AS CHAR.
DEFINE INPUT PARAMTER values AS RAW.
```

Getting Item Values By Name In a MapMessage

A MapMessage supports getting multiple named item pairs: {item-name, item-value}. The following functions convert data from JMS data types to 4GL data types.

The following function gets a boolean item from a MapMessage:

SYNTAX

```
FUNCTION getLogical RETURNS LOGICAL (itemName AS CHAR).
```

The following function gets an int, short, or bytes item from a MapMessage:

SYNTAX

```
FUNCTION getInt RETURNS INT (itemName AS CHAR).
```

The following function gets any numeric item from a MapMessage:

SYNTAX

```
FUNCTION getDecimal RETURNS DECIMAL (itemName AS CHAR).
```

The following function gets an item of any data type except bytes from a MapMessage:

SYNTAX

```
FUNCTION getChar RETURNS CHAR (itemName AS CHAR).
```

The following function gets a bytes item from a MapMessage:

SYNTAX

```
FUNCTION getBytesToRaw RETURNS RAW (itemName AS CHAR).
```

BytesMessage

A BytesMessage is an uninterpreted stream of bytes.

To create a BytesMessage, an application calls createBytesMessage in a Session object with the following output parameter:

SYNTAX

```
PROCEDURE createBytesMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.
```

The following function returns the number of bytes in a BytesMessage:

SYNTAX

```
FUNCTION getBytesCount RETURNS INT.
```

The following function returns true if the last bytes segment was retrieved:

SYNTAX

```
FUNCTION endOfStream RETURNS LOGICAL.
```

NOTE: An application should not call endOfStream if it used getMemptr to extract the data.

The following procedure changes the mode of a BytesMessage from write-only mode to read-only mode and places the cursor before the first bytes segment. Sending the message causes an implicit reset. The message arrives at the receiver in a reset state:

SYNTAX

```
PROCEDURE reset.
```

MultipartMessage

A multipart message contains one or more message parts. Each message part is identified by a content type and a content ID, and contains a set of message headers and a message body. Message parts can be Sonic messages, character data, and binary data.

When you access a multipart message, you can retrieve each part as a SonicMQ message (if that is how it was sent), as a character data, or as a byte array. The SonicMQ Adapter supports only message parts and byte-array parts returned as a CHARACTER string or a MEMPTR.

The following method creates a multipart message:

SYNTAX

```
PROCEDURE createMultipartMessage.
DEFINE OUTPUT PARAMETER messageH AS HANDLE.
```

The following method adds a Sonic message to a multipart message:

SYNTAX

```
PROCEDURE addMessagePart.
DEFINE INPUT PARAMETER messagePartH AS HANDLE.
DEFINE INPUT PARAMETER contentIDString AS CHARACTER.
```

The following method adds binary data to a multipart message:

SYNTAX

```
PROCEDURE addBytesPart.
DEFINE INPUT PARAMETER memptr AS MEMPTR.
DEFINE INPUT PARAMETER contentTypeString AS CHARACTER.
DEFINE INPUT PARAMETER contentIDString AS CHARACTER.
```

The following method adds CHARACTER data to a multipart message:

SYNTAX

```
PROCEDURE addTextPart.
DEFINE INPUT PARAMETER charString        AS CHARACTER.
DEFINE INPUT PARAMETER contentTypeString AS CHARACTER.
DEFINE INPUT PARAMETER contentIDString   AS CHARACTER.
```

The following method returns the number of parts contained by a multipart message:

SYNTAX

```
FUNCTION getPartCount RETURNS INTEGER.
```

The following method returns TRUE if the part specified by the index is a SonicMQ message:

SYNTAX

```
FUNCTION isMessagePart RETURNS LOGICAL
  (INPUT messageH AS HANDLE, INPUT index AS INTEGER).
```

The following method returns the message part corresponding to the given index:

SYNTAX

```
FUNCTION isMessagePart RETURNS CHARACTER
  (INPUT index AS INTEGER, OUTPUT messagePartH AS HANDLE).
```

The following method returns the message part corresponding to the given content ID:

SYNTAX

```
FUNCTION getMessagePartByID RETURNS CHARACTER
  (INPUT contentID AS INTEGER, OUTPUT messagePartH AS HANDLE).
```

The following method returns the bytes part corresponding to the given index:

SYNTAX

```
FUNCTION getMessagePartByID RETURNS CHARACTER
  (INPUT iIndex AS INTEGER, OUTPUT memPtr AS MEMPTR).
```

The following method returns the bytes part corresponding to the given content ID:

SYNTAX

```
FUNCTION getBytesPartByID RETURNS CHARACTER
  (INPUT-OUTPUT memPtr AS MEMPTR, INPUT contentID AS INTEGER).
```

Overcoming the 32K RAW Bytes Limit

The RAW data type has a 32K size limit. To bypass this limit, an application can do multiple read... and write... calls as with a TextMessage.

The following procedure can be called in write-only mode to write an additional bytes segment to a BytesMessage:

SYNTAX

```
PROCEDURE writeBytesFromRaw.
DEFINE INPUT PARAMETER bytesValue AS RAW.
```

The following function can be called in read-only mode to return the next bytes segment. The size of all the bytes segments other than the last one is 8192; the size of the last one is 8192 or less:

SYNTAX

```
FUNCTION readBytesToRaw RETURNS RAW.
```

Writing and Reading MEMPTR Bytes Data

The MEMPTR data type does not have a 32K limit. An application can make one get... or set... MEMPTR call.

The following procedure sets the specified number of bytes from the MEMPTR variable starting at startIndex (the first byte is 1). The setMemptr procedure implicitly calls clearBody before setting the data and resets after setting the data. Therefore, it can be used whether the message is in read-only or write-only mode prior to the call. The call makes a copy of the data. Thus, the memptrVal variable is not modified by the 4GL–JMS implementation and can be modified by the 4GL application after the call without corrupting the message:

SYNTAX

```
PROCEDURE setMemptr.
DEFINE INPUT PARAMETER memptrVar AS MEMPTR.
DEFINE INPUT PARAMETER startIndex AS INT.
DEFINE INPUT PARAMETER numBytes AS INT.
```

The following function returns a reference to a MEMPTR variable that contains exactly all the bytes of the message. The getMemptr function implicitly calls reset. If the message was in a write-only mode, it will be in a read-only/reset mode after the call. The getMemptr function does not create a copy of the MEMPTR variable; it returns a reference to the data maintained by the Message object. The deleteMessage call releases the variable's memory, and the caller must copy any data it will need or need to modify before deleting the message:

SYNTAX

```
FUNCTION getMemptr RETURNS MEMPTR.
```

Discardable Messages

Sonic contains a message delivery mode called DISCARDABLE for messages published to a topic. When a message with the delivery mode DISCARDABLE is published to a topic, if the topic's queue is full, the message is automatically deleted.

The setDefaultPersistency method sets the persistency mode of a session. Its syntax is:

SYNTAX

```
PROCEDURE setDefaultPersistency.
DEFINE INPUT PARAMETER deliveryMode AS CHAR.
```

The publish method, which publishes a message to a session, lets you specify the delivery mode. Its syntax is:

SYNTAX

```
PROCEDURE publish.
DEFINE INPUT PARAMETER topicName AS CHAR.
DEFINE INPUT PARAMETER message AS HANDLE.
DEFINE INPUT PARAMETER priority AS INT.            /*Session default is
                                                     used if UNKNOWN.*/
DEFINE INPUT PARAMETER timeToLive AS DECIMAL.      /*Session default is
                                                     used if UNKNOWN.*/
DEFINE INPUT PARAMETER deliveryMode AS CHAR.       /*Session default is
                                                     used if UNKNOWN.*/
```

XMLMessage

SonicMQ's XMLMessage is an extension of TextMessage and it supports all the methods of TextMessage. For more information, see the "TextMessage" section in this chapter.

To create an XML message, an application calls createXMLMessage in a Session object with the following output parameter:

SYNTAX

```
PROCEDURE createXMLMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.
```

XML Code-page Encoding

4GL applications work with the built-in XML parser. It is important to consider the code-page encoding of XML messages. In principle, XML documents can be encoded with any code page. However, XML parsers support some or all code pages, and XML parsers also differ in the code-page conversions they support.

4GL clients set and get XML text using the 4GL CHARACTER data type. CHARACTER data is encoded by the 4GL interpreter using the internal code page (the –cpinternal startup parameter). The 4GL–JMS implementation automatically converts the text to Unicode when sent to the JMS server and from Unicode to the internal client's code page when sent from the server to client.

In general, when the characters used by the XML document are from the 7 byte ASCII subset, there are no issues the 4GL programmer has to consider. Otherwise, observe the following examples and guidelines.

Example 1

In this example, two 4GL clients use the ISO8859-1 code page:

- Client1 sets XML text in an XML message and sends it.
- Client2 receives the message, extracts the text, stores it in a MEMPTR variable, and creates an XML document. (See the "Publishing, Receiving, and Parsing an XMLMessage" section in Chapter 14, "Using the SonicMQ Adapter.")

The following code-page conversions take place:

1. ISO8859-1 (client1) to Unicode (SonicMQ XML message)
2. Unicode (SonicMQ XML message) to ISO8859-1 (client2)

In this example, the XML parser parses the XML document correctly if the header of the document specifies that the encoding is ISO8859-1 and the parser can handle ISO8859-1.

Example 2

In this example, two 4GL clients use ISO8859–1 for their internal code page. Client1 saves a UTF–8 encoded XML document in a MEMPTR variable (using the X–DOC:SAVE() 4GL method) and then uses the 4GL GET–STRING statement to extract the text from the MEMPTR and pass it into the XML message. (This is a deliberate error.) *UTF–8* (Unicode Transformation Format) is an 8-bit encoding form that serializes a Unicode scalar value as a sequence of one to four bytes.

A 4GL client cannot mix code pages. The text it sets in the XML message must be encoded in the same code page as the client's internal code page. In general, a MEMPTR variable must be used carefully, since it can have any data in it. The 4GL programmer must be sure that it contains only NULL free text (no embedded NULL bytes), encoded with the same code page as the internal code page, before loading it into an XML message.

In this example, if the 4GL client cannot be started up with –cpinternal UTF–8, but still wants to use 4GL–JMS to pass that UTF–8 document, it can use a BytesMessage or bytes elements in a StreamMessage. When sent as bytes, the XML data will get to the receiver uninterpreted and unconverted. The 4GL receiver can then set the data in a MEMPTR variable and load the parser. (See the "Publishing, Subscribing, and Receiving an XML Document In a BytesMessage" section in Chapter 14, "Using the SonicMQ Adapter.") A second option is to convert the text (and the document's header) to ISO8859–1 using the CODEPAGE–CONVERT 4GL function.

If the 4GL receiver of an XML message is unsure about the XML header encoding declaration, it must check it and perhaps modify it to match its internal code page before loading the parser.

14

Using the SonicMQ Adapter

This chapter includes the following sections:

14.1 Installing, Configuring, and Administering the SonicMQ Adapter

The Progress SonicMQ Adapter communicates between clients and the Progress SonicMQ message broker. The SonicMQ Adapter is part of the AppServer administration framework, which also includes the Progress AdminServer, the Progress Explorer and the Progress NameServer.

For information on configuring the AdminServer and NameServer, see the chapter on configuring Progress unified broker products in the *Progress Installation and Configuration Guide Version 9 for Windows* or the *Progress Installation and Configuration Guide Version 9 for UNIX*.

NOTE: When you install the Progress AdminServer, you are asked if you want to enable user authorization. The appropriate response is not intuitively obvious. And if you do not respond appropriately, the SonicMQ Adapter might not operate as expected. For more information on installing the Progress AdminServer and enabling user authorization, see the *Progress Version 9 Product Update Bulletin*.

14.1.1 Installing the SonicMQ Adapter

Before installing the SonicMQ Adapter, confirm that you have the SonicMQ broker installed and that you have access to it. The SonicMQ Broker installation does not have to be on the same machine where the SonicMQ Adapter is to be installed.

To install the SonicMQ Adapter, follow these steps:

1 ♦ Confirm that the SonicMQ broker is installed and that you have access to it.

NOTE: The SonicMQ broker does not have to be installed on the same machine as the SonicMQ Adapter.

2 ♦ Copy the SonicMQ JAR (`.jar`) files by following these steps:

a) On the machine where you want to install the SonicMQ Adapter, create a `\sonicMQ\lib` directory under the `%DLC%\lib` directory.

b) Copy the following files from the SonicMQ broker machine's `progress-sonicmq-install-dir\lib` directory to the client machine's `\sonicMQ\lib` directory:

- `sonic_Client.jar`
- `sonic_Selector.jar`
- `sonic_Crypto.jar`
- `sonic_ASPI.jar`
- `sonic_XA.jar`
- `sonic_XMessage.jar`
- `activation.jar`

NOTE: If you upgrade your version of SonicMQ in the future, you must copy the new JAR files from the client machine's `\sonicMQ\lib` directory to the `progress-sonicmq-install-dir\lib` directory.

14.1.2 Configuring the SonicMQ Adapter With the Progress Explorer

On the Windows platform, you can use the Progress Explorer to start, stop, get status, add, delete, and edit properties of the SonicMQ Adapter. The 4GL program connects to the SonicMQ Adapter by specifying the connection parameters of the Name Server when the JMS session is created. For more information on using the Progress Explorer, see the online Help.

14.1.3 Configuring the SonicMQ Adapter From the Command Line

The SonicMQ Adapter can be configured and started by manually editing the `ubroker.properties` file. The Adapter root group in the `ubroker.properties` file supports the SonicMQ Adapter broker.

If the default service name for the SonicMQ Adapter in appserviceNameList is modified from `adapter.progress.jms`, the new value must be specified by the 4GL client using the setAdapterService Session object method specified in the property in the `ubroker.properties` file.

The following portions of the `ubroker.properties` file include the SonicMQ Adapter property groups and properties:

```
# Linkage to the parent group name for the plugin for the Unified Broker tool #
```

```
[ParentGroup]
    WebSpeed=UBroker.WS
    AppServer=UBroker.AS
    NameServer=NameServer
    Oracle DataServer=UBroker.OR
    ODBC DataServer=UBroker.OD
    MSS DataServer=UBroker.MS
    Messengers=WebSpeed.Messengers
    SonicMQ Adapter=Adapter
    AppServer Internet Adapter=AIA
```

```
 # Default properties for SonicMQ Adapter broker #
 [Adapter]
     infoVersion=9010
     workDir=@{WorkPath}
     srvrLogFile=@{WorkPath}\server.log
     brokerLogFile=@{WorkPath}\broker.log
     maxClientInstance=512
     registrationRetry=30
     userName=
     groupName=
     appserviceNameList=
     portNumber=3600
     controllingNameServer=
     environment=
     uuid=
     brkrLoggingLevel=2
     brkrLogAppend=1
     srvrLoggingLevel=2
     srvrLogAppend=1
     jvmArgs=@{JAVA\JVMARGS}
     registrationMode=Register-IP
     hostName=
     description=
     classMain=com.progress.ubroker.broker.ubroker
     srvrStartupParam=
```

```
# Sample SonicMQ adapter definition #
 [Adapter.sonicMQ1]
     srvrLogFile=@{WorkPath}\sonicMQ1.server.log
     brokerLogFile=@{WorkPath}\sonicMQ1.broker.log
     appserviceNameList=adapter.progress.jms
     uuid=932.99.999.XXX:1ee77e:cf3bbe3d33:-8030
     portNumber=3620
     controllingNameServer=NS1
     description=Sample SonicMQ adapter broker
```

You can specify the following attributes in the srvrStartupParam property of the SonicMQ Adapter. Specifying srvrStartupParam attributes is optional when a single default is desired for all of the clients. The 4GL–JMS API allows clients to overwrite the srvrStartupParam default. Names of attributes are case sensitive; the attributes must be separated with a semicolon (;):

```
jmsServerName              /*The default is SonicMQ.*/
brokerURL                  /*The default is null.*/
user                       /*The default is null.*/
password                   /*The default is null.*/
clientID                   /*The default is null.*/
pingInterval               /*In seconds; setPingInterval is not called
                             by default.*/
```

For example:

```
srvrStartupParam=brokerURL=localHost; user=u1; password=p1;
```

You can use two command-line tools, adaptconfig and adaptman, with the SonicMQ Adapter on all Progress-supported platforms.

Adaptconfig

Use adaptconfig to validate manual changes you made to the `ubroker.properties` file for SonicMQ Adapter instances:

```
adaptconfig [
  [ [ -name adapter-broker ]
    [ -propfile path-to-properties-file ]
    [ -validate ]
 ] | -help ]
```

The adaptconfig tool has these parameters:

- **–help** or **–h** — Displays command-line help
- **–name** or **–i (Name)** — Name of the SonicMQ Adapter Broker (required)
- **No options** — List all defined SonicMQ Adapter brokers
- **–f** or **–propfile (propFilePath)** — Full properties file path (optional)
- **–validate** or –v — Validate

Adaptman

Use adaptman to start, stop, query, and kill an existing SonicMQ Adapter broker. Enter the –i or the –name parameter followed by the name of the adapter broker and then the command to start, stop, query, or kill a broker. You can also use adaptman to manipulate brokers on other machines by using the –host and the –port parameters to specify the name of the machine and the port the AdminServer is running on:

```
adaptman {
  { -name adapter-broker
    { -kill | -start | -stop | -query }
    [ -host host-name -user user-name | -user user-name ]
    [ -port port-number ]
  } | -help }
```

The adaptman tool has these parameters:

- **–help** or **–h** — Displays command line help
- **–name** or **–i (Name)** — Name of the SonicMQ Adapter Broker (required)
- **–start** or **–x** — Starts the named SonicMQ Adapter broker
- **–user** or **–u (UserName)** — User name
- **–host** or **–r** — Host name where the AdminServer is running
- **–port** — Port number of the running AdminServer
- **–query** or **–q** — Queries the named SonicMQ Adapter broker
- **–kill** or **–k** — Causes emergency shutdown of the SonicMQ Adapter broker
- **–stop** or **–e** — Stops the SonicMQ Adapter broker

The following examples show how to use the adaptman parameters.

Use the following adaptman parameters to start an instance called SonicMQ1:

```
adaptman -i sonicMQ1 -start
```

Use the following adaptman parameters to query the instance for its status:

```
adaptman -i sonicMQ1 -query
```

Use the following adaptman parameters to stop an instance:

```
adaptman -name sonicMQ1 -stop
```

Use the following adaptman parameters to get status of an instance on the machine whose AdminServer is on port 12935:

```
adaptman -host xxxxxx -port 12935 -i sonicMQ1 -q
```

Use the following adaptman parameters to kill an instance:

```
adaptman -i sonicMQ1 -kill
```

14.1.4 Configuring the SonicMQ Adapter To Not Use a NameServer

You can configure the SonicMQ Adapter to not use a NameServer. To do so, use one of the following techniques:

- Configure the SonicMQ Adapter in Progress Explorer to indicate that the Adapter should not register with a NameServer.
- Connect clients directly to the SonicMQ Adapter by specifying its TCP/IP host and port. There is a new connection parameter, DirectConnect (`-DirectConnect`), that specifies that no NameServer should be used when the client connects to the Adapter.

For more information, see the *Progress Version 9 Product Update Bulletin*.

14.1.5 Configuring HTTP and HTTPS Tunneling

The SonicMQ product and the Progress SonicMQ Adapter support passing messages using HTTP and HTTPS. Although these are not the default protocols, you can configure the Sonic and Progress environments for them.

For information on configuring the SonicMQ product for HTTP and HTTPS, see the SonicMQ documentation.

To configure the Progress SonicMQ Adapter for HTTPS:

1 ♦ Shut down all instances of the Sonic broker and the Progress AdminServer.

2 ♦ Define the following settings in the Sonic `broker.ini` file in the Sonic installation directory:

```
ENABLE_SECURITY=TRUE
DEFAULT_SOCKET-TYPE=HTTPS
```

3 ♦ Rebuild the Sonic databases by entering the commands for your operating system:

Windows	UNIX
cd *sonic-install-dir* bin\dbtool /d basic bin\dbtool /cs basic bin\dbtool /c security	cd *sonic-install-dir* bin/dbtool /d basic bin/dbtool /cs basic bin/dbtool /c security

4 ♦ Modify the JVMARGS and PROGRESSCP environment variables, following the instructions for your operating system:

On Windows, modify them in the registry at the following key:

```
\\HKEY LOCAL MACHINE\SOFTWARE\PSC\PROGRESS\psc-version\JAVA
```

where *psc-version* represents the Progress version, such as 9.1D.

On UNIX, modify them at the command line.

The modifications are:

a) To the JVMARGS environment variable, append the following:

```
-DSSL_CA_CERTIFICATES_DIR=sonic-install-dir\certs\CA
```

b) To the PROGRESSCP environment variable, append the following:

```
sonic-install-dir\lib\jaxp.jar
sonic-install-dir\lib\certj.jar
sonic-install-dir\lib\sslj.jar
sonic-install-dir\lib\jsafe.jar
```

5 ♦ Modify your SonicMQ 4GL application to run setUser and setPassword before running beginSession. Doing so lets you specify the user name and password Sonic will use to authenticate users.

Sonic sets the initial value of both user name and password to Administrator. To set up additional user name-password pairs, use the Sonic Explorer. For more information, see the *SonicMQ Configuration and Administration Guide*.

6 ♦ Modify your the SonicMQ 4GL application so that each time setBrokerURL is run, the *brokerURL* input parameter specifies HTTPS.

7 ♦ Start the Sonic broker.

8 ♦ Start the Progress AdminServer.

9 ♦ Start the Progress SonicMQ Adapter.

14.2 Maximizing Performance

The primary goal of a JMS messaging system is to reliably distribute asynchronous business events and information between applications. This is achieved by a loosely coupled communication style of application integration. A more tightly coupled communication mechanism, such as sockets or direct calls to the Progress AppServer, is useful for passing large amounts of data or for subsecond response time.

14.2.1 Performance Comparison

The following example illustrates the kind of performance you can expect. It compares passing data between two 4GL clients through a JMS server with passing the same data between two 4GL clients through an AppServer application.

This configuration includes:

- Two 4GL clients on a Solaris SPARC 20 machine
- Progress SonicMQ broker on a Windows NT 300MHz machine on the LAN

The first client publishes the customer table of the Sports database as a StreamMessage with each record written as a bytes item using RAW–TRANSFER. The second client subscribes to the JMS server, receives the message, and puts the data in a temp–table. It takes, on average, 1.5 seconds to transfer the table.

Passing the customer table from one client to another through the AppServer (by passing it from the first client as an input temp–table to an AppServer application and then passing it to the second client, from the AppServer application, as an output temp–table) takes, on average, 1.3 seconds.

14.2.2 Optimizing Message Size

When performance is an issue, fewer and larger messages perform better than many small messages. The optimal message size is several thousand bytes.

14.2.3 StreamMessage, MapMessage, and TextMessage

StreamMessages and MapMessages consist of individual items (or chunks) of data. The larger the items are, the better the performance is. For example, a group of database records can be sent in a StreamMessage with each field as a separate item (using a write... method). Much better performance is achieved if each record is converted to RAW data and written as a Bytes item in a StreamMessage using writeBytesFromRaw(). Applications can use multiple appendText methods to generate larger messages in a TextMessage and XMLMessage. Using larger segments in each appendText improves performance.

14.2.4 Remote and Local Calls

In general, local 4GL–JMS API calls are less expensive than remote calls (calls that go to the SonicMQ Adapter and the SonicMQ broker). Remote procedures and functions are noted in the "Programming With the 4GL–JMS API" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL" and in the "4GL–JMS API Alphabetical Reference" section in Appendix C, "4GL–JMS API Reference." This information is useful when analyzing the performance of an application.

14.2.5 Message Reuse

The creation of a 4GL message is relatively expensive. The publisher (or sender) of a message should reuse a Message object whenever possible. The message can be cleared for reuse by calling clearBody and clearProperties. The message body of some message types is automatically cleared when new data is set. (For more information, see the "clearBody and clearProperties" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

The application that consumes messages can reuse them by calling the setReuseMessage message consumer method. If setReuseMessage is called, the message consumer reuses the same Message object for all the messages it receives, provided that the message was not deleted by the application.

14.3 Load Balancing

SonicMQ supports client-side load balancing. With this enabled, a connect request can be redirected to another broker within the SonicMQ cluster, provided broker-side load balancing has not been disabled.

Client-side load balancing involves the following methods on the session handle:

- setLoadBalancing
- getLoadBalancing

For more information on load balancing, see the reference entries for these methods in Appendix C, “4GL–JMS API Reference.”.

14.3.1 Discardable Messages

When you publish a message to a topic, you can specify the DISCARDABLE delivery mode. If you do and the destination message queue is full, the message is automatically discarded.

You can specify the DISCARDABLE delivery mode in the following methods:

- setDefaultPersistency
- publish

For more information on discardable messages, see the reference entries for these methods in Appendix C, “4GL–JMS API Reference.”.

14.4 Finding Administered Objects In JNDI Or Proprietary Directories

A JMS *administered object* is an object created by a JMS administrator and registered with a directory (typically a JNDI-compliant directory) under a name that is meaningful to the JMS clients. The object contains JMS configuration information that is created by a JMS administrator and later used by JMS clients. *JNDI* (Java Naming and Directory Interface) is an interface for JMS administrators to create and configure administered objects and store them in a namespace.

The SonicMQ-administered objects are:

- TopicConnectionFactory
- QueueConnectionFactory
- Topic
- Queue

For example, the administrator creates a TopicConnectionFactory object, which contains all the JMS server connection parameters (communication protocol host and port), assigns it a name, and stores it in a JNDI directory. The client does not have to know the connection parameters to connect to the JMS server. The client finds the object by name in the directory and uses it to create connection objects. The administrator can change the connection parameters later without affecting client applications.

The administrator can give the Topic and Queue objects meaningful aliases to shield the client from their internal names. For example, a topic with the internal JMS name of sports.USA.Northeast.golf could be stored in the directory under “northern.golfers.” For more information on administered objects, see the *Java Message Service* specification and the *SonicMQ Programming Guide*.

14.4.1 Using the SonicMQ Adapter and the 4GL–JMS API With Administered Objects

JMS does not impose any specific directory for storing administered objects (although it establishes the convention of using JNDI-compliant directories, such as LDAP). Also, the process of connecting to a JNDI server and obtaining an initial context is not standardized.

Therefore, to use directory-stored JMS objects, you must implement a Java class, compile it, and install the class file on the SonicMQ Adapter host under the SonicMQ Adapter’s CLASSPATH. The SonicMQ Adapter looks for that class when it starts up. If it finds the class, it creates an instance object of it and uses it to locate administered objects. If it does not find the class, the SonicMQ Adapter creates objects as required.

jmsfrom4gl.AdminObjectFinder Class

The following code is the skeleton of the jmsfrom4gl.AdminObjectFinder class. Use it as a template to create a class file and install it on the SonicMQ Adapter host. The jmsfrom4gl.AdminObjectFinder name is mandatory. The class and the get...() methods must be declared public. The AdminObjectFinder class must be part of the jmsfrom4gl package and placed in a directory called `jmsfrom4gl`. The directory that contains `jmsfrom4gl` must be on the CLASSPATH of the Ubroker:

```
package jmsfrom4gl;
import javax.jms.TopicConnectionFactory;
import javax.jms.QueueConnectionFactory;
import javax.jms.Topic;
import javax.jms.Queue;

public class AdminObjectFinder
{
  public TopicConnectionFactory getTopicConnectionFactory(String name)
    throws Exception
  {
    TopicConnectionFactory factory = null;
    // Write code to populate factory
    return factory;
  }
  public QueueConnectionFactory getQueueConnectionFactory(String name)
    throws Exception
  {
    QueueConnectionFactory factory = null;
    // Write code to populate factory
    return factory;
  }
  public Topic getTopic(String name)
    throws Exception
  {
    Topic topic = null;
    // Write code to populate topic
    return topic;
  }
  public Queue getQueue(String name)
    throws Exception
  {
    Queue queue = null;
    // Write code to populate queue
    return queue;
  }
}
```

The following sections explain how to set the CLASSPATH by modifying the value for PROGRESSCP.

Setting the CLASSPATH On Windows

On Windows, you can set the CLASSPATH by using the Progress in2reg utility to modify the value for the PROGRESSCP environment variable.

Follow these steps to modify the PROGRESSCP environment variable:

1 ♦ Modify the [JAVA] section of the `progress.ini` file (in *<Progress_install_dir>*`\bin`) by appending a semicolon followed by the full path to the `\jmsfrom4gl` directory at the end of the current information.

2 ♦ Run ini2reg on the `progress.ini` file, changing the registry base key to HKEY_LOCAL_MACHINE.

For more information on the ini2reg utility, see its online help.

Setting the CLASSPATH On UNIX

On UNIX, you can set the CLASSPATH by editing the `java_env` script (in the *<install-directory>*`/bin` directory). The `java_env` script contains a definition for the PROGRESSCP environment variable. Append a colon followed by the full path to the `/jmsfrom4gl` directory at the end of the entry for PROGRESSCP.

NOTES:

- The brokerURL startup parameter is used as the input parameter for the getTopicConnectionFactory and getQueueConnectionFactory methods. For example, if the 4GL application calls setBrokerURL(“directory_factory_name”), the 4GL–JMS implementation on the server side calls the getTopicConnectionFactory method with “directory_factory_name” as the parameter.

- If the getTopicConnectionFactory and getQueueConnectionFactory methods are implemented, the jmsServerName startup parameter is ignored (since the identity of the server’s vendor is encapsulated in the object).

- It is sufficient to implement methods for those objects that should be obtained from the directory. For example, it is legal to have an AdminObjectFinder class with only the getTopicConnectionFactory method. The 4GL–JMS implementation looks for the methods dynamically and does not fail if the other methods are missing.

- If the object finder method returns null, the 4GL–JMS implementation tries to create the object as if the method is not there.

14.5 Internationalization Considerations

The 4GL interpreter (for the client, AppServer, and WebSpeed) supports many code-page encoding standards. The JMS client uses Unicode. The translation of text data between the 4GL's code page and Unicode is done automatically by the 4GL–JMS implementation. (For more information, see the "XML Code-page Encoding" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL.")

When a 4GL client sends text data to JMS (for example, in a TextMessage or a StreamMessage), the 4GL client must send the text in a Unicode/utf-8 format. If the internal code page of the client is not in Unicode/UTF–8 format (–cpinternal UTF–8), the 4GL–JMS implementation must convert the text to UTF–8.

When text is converted to UTF–8, each character can require up to three bytes. This causes the text size limit of each text chunk to be 10K, since the conversion routine must prepare enough expansion room. Since all the message types support segmentation of text data, the limit can be worked around by using multiple segments. Whenever possible, the 4GL client's internal code page should be set to UTF–8 to avoid performing code-page conversions and to eliminate the 10K size limit.

14.6 Running the Messaging Examples and the Gateway Sample Application

Progress includes 4GL code examples of Pub/Sub and PTP messaging as a well as a sample application illustrating the gateway approach to integration with the native 4GL publish and subscribe mechanism.

14.6.1 Pub/Sub Messaging Examples

The Pub/Sub examples consist of sets of subscribers and publishers. You should run each messaging example interactively from its own window. Launch the subscriber first, because the message is discarded if the publisher publishes the message before there are any subscribers to the topic (or any durable subscriptions).

The path for the messaging examples depends on your operating system. The paths are:

Windows	UNIX
%DLC%\src\samples\sonicmq\adapter\examples\example*.p	$DLC\src\samples\sonicmq\adapter\examples\example*.p

Publishing and Subscribing With a TextMessage

Examples 1 and 2 demonstrate basic Pub/Sub messaging. Example 1 publishes a TextMessage to a topic and Example 2 subscribes to a topic and receives a TextMessage.

Follow these steps to run Examples 1 and 2:

1 ♦ Run `example2.p` so the subscriber is running before you publish.

example2.p *(1 of 2)*

```
/* Subscribes and receives a Text message. */
DEFINE VARIABLE pubsubsession AS HANDLE.
DEFINE VARIABLE consumerH AS HANDLE.

/* Creates a session object. */
RUN jms/pubsubsession.p PERSISTENT SET pubsubsession("-H localhost -S
5162 ").
RUN setBrokerURL IN pubsubsession ("localhost:2506").
RUN beginSession IN pubsubsession.

/* Subscribe to the GolfTopic topic. Messages are handled by the
   "golfHandler" internal procedure.
*/
RUN createMessageConsumer IN pubsubsession (
                  THIS-PROCEDURE, /* This proc will handle it */
                  "golfHandler",  /* name of internal procedure */
                  OUTPUT consumerH).

RUN subscribe IN pubsubsession (
              "GolfTopic",     /* name of topic */
              ?,               /* Subscription is not durable */
              ?,               /* No message selector */
              no,              /* Want my own messages too */
              consumerH).      /* Handles the incoming messages*/

/* Start receiving messages */
RUN startReceiveMessages IN pubsubsession.

/* Wait to receive the messages. Any other IO-blocked statements can be
   used for receiving messages.
*/
WAIT-FOR u1 OF THIS-PROCEDURE.
```

example2.p *(2 of 2)*

```
PROCEDURE golfHandler:
DEFINE INPUT PARAMETER messageH AS HANDLE.
DEFINE INPUT PARAMETER msgConsumerH AS HANDLE.
DEFINE OUTPUT PARAMETER replyH AS HANDLE.

/* Display the message - we assume that reply is not required. */
DISPLAY "Message text: " DYNAMIC-FUNCTION('getText':U IN messageH)
  FORMAT "x(70)".
RUN deleteMessage IN messageH.

APPLY "U1" TO THIS-PROCEDURE.
END.
```

2 ♦ Run example1.p to publish the TextMessage to a topic.

example1.p

```
/* Publishes A Text message. */
DEFINE VARIABLE pubsubsession AS HANDLE.
DEFINE VARIABLE messageH AS HANDLE.

/* Creates a session object. */
RUN  jms/pubsubsession.p PERSISTENT SET pubsubsession ("-H localhost -S
5162 ").
RUN setBrokerURL IN pubsubsession ("localhost:2506").
RUN beginSession IN pubsubsession.

/* Create a text message */
RUN createTextMessage IN pubsubsession (OUTPUT messageH).
RUN setText IN messageH ("Golf shoes on sale today").

/* Publish the message on the "GolfTopic" topic */
RUN publish IN pubsubsession ("GolfTopic", messageH, ?, ?, ?).

RUN deleteMessage IN messageH.
RUN deleteSession IN pubsubsession.
```

Publishing With Message Properties and Subscribing Selectively

Example 3 publishes a TextMessage from Super Golf Center to Sub Par Golf using setStringProperty. Example 4 subscribes to a topic and only receives messages addressed to “Sub Par Golf” (by passing a selector to the Subscribe call).

Follow these steps to run Examples 3 and 4:

1 ♦ Run example4.p first so the subscriber is running before you publish.

example4.p *(1 of 2)*

```
/* Receives a Text message with the "TO" property equals to "Sub Par Golf"
*/
DEFINE VARIABLE pubsubsession AS HANDLE.
DEFINE VARIABLE consumerH AS HANDLE.

/* Creates a session object. */
RUN  jms/pubsubsession.p PERSISTENT SET pubsubsession ("-H localhost -S
5162 ").
RUN setBrokerURL IN pubsubsession ("localhost:2506").
RUN beginSession IN pubsubsession.

/* Subscribes to the GolfTopic topic. Messages are handled by the
   "golfHandler" internal procedure.  */
RUN createMessageConsumer IN pubsubsession (
                     THIS-PROCEDURE,    /* This proc will handle it */
                     "golfHandler",     /* name of internal procedure */
                     OUTPUT consumerH).

RUN subscribe IN pubsubsession (
             "GolfTopic",           /* name of topic */
             ?,                     /* Subscription is not durable */
             "TO = 'Sub Par Golf'", /* Only messages from Sub Par Golf */
             no,                    /* Want my own messages too */
             consumerH).            /* Handles the incoming messages */

/* Start receiving messages */
RUN startReceiveMessages IN pubsubsession.

/* Wait to receive the messages. Any other IO-blocked statements can be
   used for receiving messages.
*/
WAIT-FOR u1 OF THIS-PROCEDURE.
```

example4.p *(2 of 2)*

```
PROCEDURE golfHandler:
DEFINE INPUT PARAMETER messageH AS HANDLE.
DEFINE INPUT PARAMETER msgConsumerH AS HANDLE.
DEFINE OUTPUT PARAMETER replyH AS HANDLE.

/* Display the message - we assume that reply is not required. */
DISPLAY "Message text: " DYNAMIC-FUNCTION('getText':U IN messageH)
                         FORMAT "X(30)"
      "Message from: " DYNAMIC-FUNCTION('getCharProperty':U IN messageH,
                                        "FROM").
RUN deleteMessage IN messageH.
APPLY "U1" TO THIS-PROCEDURE.
END.
```

2 ♦ Run example3.p.

example3.p

```
/* Publishes a Text message with properties. */
DEFINE VARIABLE pubsubsession AS HANDLE.
DEFINE VARIABLE messageH AS HANDLE.

/* Creates a session object. */
RUN  jms/pubsubsession.p PERSISTENT SET pubsubsession ("-H localhost -S
5162 ").
RUN setBrokerURL IN pubsubsession ("localhost:2506").
RUN beginSession IN pubsubsession.

/* Create a text message */
RUN createTextMessage IN pubsubsession (OUTPUT messageH).
RUN setText IN messageH ("Golf shoes on sale today.").

/* Set the "FROM:" and the "TO:" properties */
RUN setStringProperty IN messageH ("FROM", "Super Golf Center").
RUN setStringProperty IN messageH ("TO", "Sub Par Golf").

/* Publish the message on to the golf topic */
RUN publish IN pubsubsession ("GolfTopic", messageH, ?, ?, ?).
```

Publishing With a Reply Handle, Subscribing, and Receiving an Automatic Reply

Examples 5, 6, and 7 illustrate publishing with a reply handle and receiving an automatic reply. Example 5 subscribes with an automatic reply mechanism. It can only reply to messages that have the JMSReplyTo header field. Example 6 subscribes with explicit reply by calling publish directly. It can only reply to messages that have the JMSReplyTo header field. Example 7 publishes using requestReply for receiving reply messages from subscribers. It populates the JMSReplyTo header field automatically.

Follow these steps to run Examples 5, 6, and 7:

1 ♦ Run example5.p so the subscriber is running before you publish.

example5.p *(1 of 2)*

```
/* Using the automatic reply mechanism. Note that the received message
   must have a JMSReplyTo header field for this to work. Example7 can be
   used to receive the reply.  */
DEFINE VARIABLE pubsubsession AS HANDLE.
DEFINE VARIABLE consumerH AS HANDLE.

/* Creates a session object. */
RUN  jms/pubsubsession.p PERSISTENT SET pubsubsession ("-H localhost -S
5162 ").
RUN setBrokerURL IN pubsubsession ("localhost:2506").
RUN beginSession IN pubsubsession.

/* Subscribe to the GolfTopic topic. Messages are handled by the
   "golfHandler" internal procedure.  */
RUN createMessageConsumer IN pubsubsession (
                      THIS-PROCEDURE,    /* This proc will handle it */
                      "golfHandler",     /* name of internal procedure */
                      OUTPUT consumerH).

RUN subscribe IN pubsubsession (
                         "GolfTopic",  /* name of topic */
                         ?,            /* Subscription is not durable */
                         ?,            /* No message selector */
                         no,           /* Want my own messages too */
                         consumerH).   /* Handles the messages */

/* Start receiving messages */
RUN startReceiveMessages IN pubsubsession.

/* Wait forever to receive messages since "u1" is never applied.  */
WAIT-FOR u1 OF THIS-PROCEDURE.
```

example5.p *(2 of 2)*

```
PROCEDURE golfHandler:
DEFINE INPUT PARAMETER messageH AS HANDLE.
DEFINE INPUT PARAMETER msgConsumerH AS HANDLE.
DEFINE OUTPUT PARAMETER replyH AS HANDLE.

/* Creates a reply message. The reply is published automatically when
   control returns to the 4GL-JMS implementation.
*/
DISPLAY DYNAMIC-FUNCTION('getText':U IN messageH) format "x(60)".
IF DYNAMIC-FUNCTION('hasReplyTo':U IN messageH) THEN
  DO:
    RUN createTextMessage IN pubsubsession (OUTPUT replyH).
    RUN setText IN replyH ("Will bid. Send data in sportsXML format.").
  END.
RUN deleteMessage IN messageH.

END.
```

2 ♦ Run example6.p to subscribe with explicit reply by calling publish directly.

example6.p *(1 of 2)*

```
/* Replying explicitly. Note that the received message must
   have a JMSReplyTo header field for this to work. Example7 can be used
   to receive the reply.  */
DEFINE VARIABLE pubsubsession AS HANDLE.
DEFINE VARIABLE msgConsumer AS HANDLE.

/* Creates a session object. */
RUN  jms/pubsubsession.p PERSISTENT SET pubsubsession ("-H localhost -S
5162 ").
RUN setBrokerURL IN pubsubsession ("localhost:2506").
RUN beginSession IN pubsubsession.

RUN createMessageConsumer IN pubsubsession (
                       THIS-PROCEDURE,   /* This proc will handle it */
                       "messageHandler", /* name of internal procedure */
                       OUTPUT msgConsumer).

RUN subscribe IN pubsubsession (
                                "GolfTopic",
                                ?,  /* No durable subscription */
                                ?,  /* No message selector */
                               no, /* Want to get my own publications */
                               msgConsumer).
RUN startReceiveMessages IN pubsubsession.

/* Wait forever to receive messages since "u1" is never applied.  */
WAIT-FOR u1 OF THIS-PROCEDURE.

RUN deleteSession IN pubsubsession.
```

example6.p *(2 of 2)*

```
PROCEDURE messageHandler:
DEFINE INPUT PARAMETER messageH AS HANDLE NO-UNDO.
DEFINE INPUT PARAMETER messageConsumerH AS HANDLE NO-UNDO.
DEFINE OUTPUT PARAMETER autoReplyH AS HANDLE NO-UNDO.
  /* Not used in this example */

DEFINE VARIABLE replyH AS HANDLE.

DISPLAY DYNAMIC-FUNCTION('getText':U IN messageH) format "x(60)".
IF NOT DYNAMIC-FUNCTION('hasReplyTo':U IN messageH) THEN RETURN.

/* Publishes a reply explicitly - using the publish call. */
RUN createTextMessage IN pubsubsession (OUTPUT replyH).
RUN setText IN replyH("Will bid. Send data in sportsXML format.").
RUN publish IN pubsubsession (DYNAMIC-FUNCTION('getJMSReplyTo':U IN
  messageH), replyH, ?, ?, ?).
RUN deleteMessage IN messageH.

END.
```

3 ♦ Run example7.p to publish using requestReply for receiving reply messages from subscribers. It populates the JMSReplyTo header field automatically.

example7.p *(1 of 2)*

```
/* Publishes a message and receives a reply. */
DEFINE VARIABLE pubsubsession AS HANDLE.
DEFINE VARIABLE consumerH AS HANDLE.
DEFINE VARIABLE messageH AS HANDLE.

/* Creates a session object. */
RUN  jms/pubsubsession.p PERSISTENT SET pubsubsession ("-H localhost -S
5162 ").
RUN setBrokerURL IN pubsubsession ("localhost:2506").
RUN beginSession IN pubsubsession.

/* Start receiving messages */
RUN startReceiveMessages IN pubsubsession.

/* Create a text message */
RUN createTextMessage IN pubsubsession (OUTPUT messageH).
RUN setText IN messageH ("Golf shoes on sale today.").

/* Creates a consumer for the reply  */
RUN createMessageConsumer IN pubsubsession (
                      THIS-PROCEDURE, /* This proc will handle it */
                      "golfHandler",  /* name of internal procedure */
                      OUTPUT consumerH).

/* Publish the message onto the Golf topic. Handle the reply in the
   golfHandler internal procedure.
*/
RUN requestReply IN pubsubsession (
                            "GolfTopic",
                             messageH,
                             ?,           /* No reply selector. */
                             consumerH, ?, ?, ?).

/* Wait forever to receive messages since "u1" is never applied.  */
WAIT-FOR u1 OF THIS-PROCEDURE.
```

example7.p *(2 of 2)*

```
PROCEDURE golfHandler:
DEFINE INPUT PARAMETER replyH AS HANDLE.
DEFINE INPUT PARAMETER msgConsumerH AS HANDLE.
DEFINE OUTPUT PARAMETER responseH AS HANDLE.

/* Display the reply - we are not sending a response. */
DISPLAY "reply text: " DYNAMIC-FUNCTION('getText':U IN replyH)
  FORMAT "X(30)".
RUN deleteMessage IN replyH.

END.
```

Publishing, Receiving, and Processing a StreamMessage

Examples 8 and 9 publish, receive, and process a StreamMessage.

Follow these steps to run Examples 8 and 9:

1 ♦ Connect to the Sports database.

2 ♦ Run example9.p to receive the StreamMessage containing the customer names and numbers.

example9.p *(1 of 2)*

```
/* Receives a Stream message. */
DEFINE VARIABLE pubsubsession AS HANDLE.
DEFINE VARIABLE consumerH AS HANDLE.

/* Creates a session object. */
RUN  jms/pubsubsession.p PERSISTENT SET pubsubsession ("-H localhost -S
5162 ").
RUN setBrokerURL IN pubsubsession ("localhost:2506").
RUN beginSession IN pubsubsession.

/* Subscribe to the newCustomers topic. The newCustHandler internal
   procedure handles the list of new customers.
*/
RUN createMessageConsumer IN pubsubsession (
                    THIS-PROCEDURE,    /* This proc will handle it */
                    "newCustHandler",  /* name of internal procedure */
                    OUTPUT consumerH).

RUN subscribe IN pubsubsession (
                     "NewCustomers", /* name of topic */
                      ?,             /* Subscription is not durable*/
                      ?,             /* No message selector. */
                      no,            /* Want my own messages too */
                      consumerH).    /* handles the messages */

/* Start receiving messages */
RUN startReceiveMessages IN pubsubsession.

/* Wait to receive the messages. Any other IO-blocked statements can be
   used for receiving messages.
*/
WAIT-FOR u1 OF THIS-PROCEDURE.
```

example9.p *(2 of 2)*

```
PROCEDURE newCustHandler:
DEFINE INPUT PARAMETER messageH AS HANDLE.
DEFINE INPUT PARAMETER msgConsumerH AS HANDLE.
DEFINE OUTPUT PARAMETER replyH AS HANDLE.

/* Display the stream of customer names and customer numbers. */
/* The moveToNext function moves the cursor to the next item in */
/* the stream and returns the data type of that item.          */
/* We assume the reply is not required. */
IF NOT  DYNAMIC-FUNCTION('getMessageType':U IN messageH) =
"StreamMessage"
     THEN RETURN.

/* Note that the 'moveToNext' functions returns the item's data type. */
 DO WHILE NOT  DYNAMIC-FUNCTION('endOfStream':U IN messageH) WITH DOWN:
    DISPLAY DYNAMIC-FUNCTION('moveToNext':U IN messageH)
            DYNAMIC-FUNCTION('readChar':U IN messageH)
            DYNAMIC-FUNCTION('moveToNext':U IN messageH)
            DYNAMIC-FUNCTION('readInt':U IN messageH).
    DOWN.
 END.
RUN deleteMessage IN messageH.
APPLY "U1" TO THIS-PROCEDURE.
END. /* newCustHandler */
```

3 ♦ Run example8.p to publish a StreamMessage containing customer names and numbers.

example8.p

```
/* Publishing a Stream message. */
DEFINE VARIABLE pubsubsession AS HANDLE.
DEFINE VARIABLE messageH AS HANDLE.

/* Creates a session object. */
RUN  jms/pubsubsession.p PERSISTENT SET pubsubsession ("-H localhost -S
5162 ").
RUN setBrokerURL IN pubsubsession ("localhost:2506").
RUN beginSession IN pubsubsession.

/* Create a stream message */
RUN createStreamMessage IN pubsubsession (OUTPUT messageH).

/* Load the message with a list of customer names and cust-nums. */
FOR EACH customer:
   RUN writeString IN messageH (customer.name).
   RUN writeInt IN messageH (customer.cust-num).
END.

/* Publish the message on the NewCustomers topic. */
RUN publish IN pubsubsession ("NewCustomers", messageH, ?, ?, ?).
```

Publishing, Receiving, and Parsing an XMLMessage

Examples 10 and 11 create, publish, receive, and parse an XMLMessage.

Follow these steps to run Examples 10 and 11:

1 ♦ Run example11.p so the subscriber is running before you publish.This example subscribes, receives, and parses an XML message.

example11.p *(1 of 3)*

```
/* Receives and parse an XML message. */
DEFINE VARIABLE pubsubsession AS HANDLE.
DEFINE VARIABLE msgConsumer1 AS HANDLE.
DEFINE VARIABLE stillWaiting AS LOGICAL INIT yes.
DEFINE VARIABLE msgNum AS INT INIT 0.

RUN  jms/pubsubsession.p PERSISTENT SET pubsubsession ("-H localhost -S
5162 ").
RUN setBrokerURL IN pubsubsession ("localhost:2506").
RUN beginSession IN pubsubsession.
RUN createMessageConsumer IN pubsubsession
      (THIS-PROCEDURE, "messageHandler", OUTPUT msgConsumer1).
RUN subscribe IN pubsubsession ("people",
                                ?,
                                ?,
                                no,
                                msgConsumer1) NO-ERROR.
RUN startReceiveMessages IN pubsubsession.
RUN waitForMessages IN pubsubsession ("inWait", THIS-PROCEDURE, ?).
RUN deleteSession IN pubsubsession.
```

example11.p *(2 of 3)*

```
PROCEDURE messageHandler:
DEFINE INPUT PARAMETER messageH AS HANDLE NO-UNDO.
DEFINE INPUT PARAMETER messageConsumerH AS HANDLE NO-UNDO.
DEFINE OUTPUT PARAMETER replyH AS HANDLE NO-UNDO.

DEFINE VAR memptrDoc AS MEMPTR.
DEFINE VAR hdoc AS HANDLE.
DEFINE VAR hRoot AS HANDLE.
DEFINE VAR xmlText AS CHAR.
DEFINE VAR indx AS INT.

CREATE X-DOCUMENT hdoc.
CREATE X-NODEREF hRoot.
SET-SIZE(memptrDoc) = 400000. /* The size is an estimate. */

indx = 1.
DO WHILE NOT  DYNAMIC-FUNCTION('endOfStream' IN messageH):
  xmlText = DYNAMIC-FUNCTION('getTextSegment':U IN messageH).
  PUT-STRING(memptrDoc, indx)  =  xmlText.
  indx = indx + LENGTH(xmlText).
END.

hdoc:LOAD("memptr", memptrDoc, FALSE).
hdoc:GET-DOCUMENT-ELEMENT(hRoot).
RUN getPeople(hRoot, 1).
RUN deleteMessage IN messageH.
SET-SIZE(memptrDoc) = 0.
stillWaiting = false.
END.
```

```
/* Displays the XML node names and XML text. */
PROCEDURE getPeople:
DEFINE INPUT PARAMETER hParent AS HANDLE.
DEFINE INPUT PARAMETER level AS INT.
DEFINE VAR i AS INT.
DEFINE VAR  hNoderef AS HANDLE.

CREATE X-NODEREF hNoderef.
REPEAT i = 1 TO hParent:NUM-CHILDREN.
      hParent:GET-CHILD(hNoderef,i).
      IF hNoderef:NAME = "#text" THEN
         MESSAGE "Text: " hNoderef:NODE-VALUE.
      ELSE
         MESSAGE "Node name: " hNoderef:NAME.
      RUN getPeople(hNoderef, (level + 1)).
END.
DELETE OBJECT hNoderef.
END.
```

example11.p *(3 of 3)*

```
FUNCTION inWait RETURNS LOGICAL.
RETURN stillWaiting.
END.
```

2 ♦ Run example10.p to create and publish an XML message with the data of 100 people.

example10.p

```
/* Publishes an XML message. */
DEFINE VARIABLE pubsubsession AS HANDLE.
DEFINE VARIABLE mesgH AS HANDLE.
DEFINE VARIABLE person AS CHAR INIT "".
DEFINE VARIABLE i AS INT.

RUN  jms/pubsubsession.p PERSISTENT SET pubsubsession ("-H localhost -S
5162 ").
RUN setBrokerURL IN pubsubsession ("localhost:2506").
RUN beginSession IN pubsubsession.
RUN createXMLMessage IN pubsubsession (OUTPUT mesgH).
```

```
/* Creates an XML message with 100 people. */
RUN appendText IN mesgH('<?xml version="1.0" ').
RUN appendText IN mesgH("encoding='ISO8859-1' ?>").
RUN appendText IN mesgH("<personnel>").
REPEAT i = 1 TO 100:
  person =            "<person>".
  person = person +     "<name>".
  person = person +         "<family>SecondName</family>".
  person = person +        "<given>FirstName" + STRING(i) +  "</given>".
  person = person +      "</name>".
  person = person +     "<email>myEmail@subpargolf.com</email>".
  person = person +  "</person>".
  RUN appendText IN mesgH(person).
END.
RUN appendText IN mesgH("</personnel>").

RUN publish IN pubsubsession ("people", mesgH, ?, ?, ?).
RUN deleteMessage IN mesgH.
RUN deleteSession IN pubsubsession.
```

Publishing, Subscribing, and Receiving an XML Document In a BytesMessage

Examples 12 and 13 use a MEMPTR variable to publish and receive an XML document in a BytesMessage to prevent code-page conversions. The code pages of the document and the 4GL client do not have to match.

Follow these steps to run Examples 12 and 13:

1 ♦ Run `example13.p` to subscribe and receive a BytesMessage containing an XML document.

example13.p *(1 of 2)*

```
/* Receives an XML document in a Bytes message. */
DEFINE VARIABLE pubsubsession AS HANDLE.
DEFINE VARIABLE msgConsumer1 AS HANDLE.
DEFINE VARIABLE stillWaiting AS LOGICAL INIT yes.

RUN  jms/pubsubsession.p PERSISTENT SET pubsubsession ("-H localhost -S
5162 ").
RUN setBrokerURL IN pubsubsession ("localhost:2506").
RUN beginSession IN pubsubsession.
RUN createMessageConsumer IN pubsubsession(THIS-PROCEDURE,
  "messageHandler",
  OUTPUT msgConsumer1).
RUN subscribe IN pubsubsession (
                    "xmlTopic",
                    ?,          /* Not a durable subscription */
                    ?,          /* No message selector. */
                    no,         /* noLocal */
                    msgConsumer1) NO-ERROR.

RUN startReceiveMessages IN pubsubsession.
RUN waitForMessages IN pubsubsession ("inWait", THIS-PROCEDURE, ?).
RUN deleteSession IN pubsubsession.
```

example13.p *(2 of 2)*

```
PROCEDURE messageHandler:
DEFINE INPUT PARAMETER messageH AS HANDLE NO-UNDO.
DEFINE INPUT PARAMETER messageConsumerH AS HANDLE NO-UNDO.
DEFINE OUTPUT PARAMETER replyH AS HANDLE NO-UNDO.

DEFINE VAR memptrDoc AS MEMPTR.
DEFINE VAR hdoc AS HANDLE.
DEFINE VAR hRoot AS HANDLE.

memptrDoc = DYNAMIC-FUNCTION('getMemptr':U IN messageH).
CREATE X-DOCUMENT hdoc.
CREATE X-NODEREF hRoot.
hdoc:LOAD("memptr", memptrDoc, FALSE).
hdoc:GET-DOCUMENT-ELEMENT(hRoot).
RUN GetChildren(hRoot, 1).
RUN deleteMessage IN messageH.
stillWaiting = false.
END.
```

```
PROCEDURE GetChildren:
DEFINE INPUT PARAMETER hParent AS HANDLE.
DEFINE INPUT PARAMETER level AS INT.
DEFINE VAR i AS INT.
DEFINE VAR  hNoderef AS HANDLE.

CREATE X-NODEREF hNoderef.
REPEAT i = 1 TO hParent:NUM-CHILDREN.
      hParent:GET-CHILD(hNoderef,i).
      IF hNoderef:NAME = "#text" THEN
         MESSAGE "Node text: " hNoderef:NODE-VALUE.
      ELSE
         MESSAGE "Node name: " hNoderef:NAME.
      RUN GetChildren(hNoderef, (level + 1)).
END.
DELETE OBJECT hNoderef.
END.
```

```
FUNCTION inWait RETURNS LOGICAL.
RETURN stillWaiting.
END.
```

2 ♦ Run example12.p to publish the BytesMessage containing an XML document.

example12.p

```
/* Publishes an XML document in a Bytes message. */
DEFINE VARIABLE pubsubsession AS HANDLE.
DEFINE VARIABLE mesgH AS HANDLE.
DEFINE VARIABLE memVal AS MEMPTR.
DEFINE VAR hdoc AS HANDLE.

RUN  jms/pubsubsession.p PERSISTENT SET pubsubsession ("-H localhost -S
5162 ").
RUN setBrokerURL IN pubsubsession ("localhost:2506").
RUN beginSession IN pubsubsession.
RUN createBytesMessage IN pubsubsession (OUTPUT mesgH).

CREATE X-DOCUMENT hdoc.
hdoc:LOAD("file", "personal.xml", FALSE).
hdoc:SAVE("memptr", memVal).

RUN setMemptr IN mesgH(memVal, ?, ?).
RUN publish IN pubsubsession ("xmlTopic", mesgH, ?, ?, ?).

SET-SIZE(memVal) = 0.
RUN deleteMessage IN mesgH.
RUN deleteSession IN pubsubsession.
```

```
/*
 The personal.xml document:

<?xml version="1.0" encoding='UTF-8' ?>
<personnel>
  <person id="Irving.Nigrini">
    <name><family>Nigrini</family> <given>Irving</given></name>
    <email>inigrini@subpargolf.com</email>
    <link manager="Thomas.Roy"/>
  </person>
  <person id="Jules.Nigrini">
    <name><family>Nigrini</family> <given>Jules</given></name>
    <email>jnigrini@subpargolf.com</email>
    <link manager="Thomas.Roy"/>
  </person>
</personnel>

*/
```

Publishing, Subscribing, and Receiving the Customer Table In a StreamMessage

Examples 14 and 15 use RAW transfer to publish, subscribe, and receive the customer table in a StreamMessage. Example 14 publishes the customer table in a StreamMessage; each customer record is a bytes item. Example 15 subscribes and receives the customer table in a StreamMessage; each customer record is a bytes item.

Follow these steps to run Examples 14 and 15:

1 ♦ Start a server for the Sports database. Each client must connect to the database in multi-user mode.

2 ♦ Run example15.p so the subscriber is running before you publish.

example15.p *(1 of 2)*

```
/* Receives the customer table in a Stream message. */
DEFINE VARIABLE pubsubsession AS HANDLE.
DEFINE VARIABLE msgConsumer1 AS HANDLE.
DEFINE TEMP-TABLE custt LIKE customer.

RUN  jms/pubsubsession.p PERSISTENT SET pubsubsession
 ("-H localhost -S 5162 ").
RUN setBrokerURL IN pubsubsession ("localhost:2506").
RUN beginSession IN pubsubsession.
RUN createMessageConsumer IN pubsubsession(THIS-PROCEDURE,
"messageHandler",
                                           OUTPUT msgConsumer1).
RUN subscribe IN pubsubsession (
                  "topic1",
                  ?,      /* Not a durable subscription */
                  ?,      /* No message selector */
                  no,     /* noLocal */
                  msgConsumer1).
RUN startReceiveMessages IN pubsubsession.
WAIT-FOR u1 OF THIS-PROCEDURE.

FOR EACH custt:
    DISPLAY custt with 2 column.
END.

RUN deleteSession IN pubsubsession.
```

example15.p *(2 of 2)*

```
PROCEDURE messageHandler:
DEFINE INPUT PARAMETER messageH AS HANDLE NO-UNDO.
DEFINE INPUT PARAMETER messageConsumerH AS HANDLE NO-UNDO.
DEFINE OUTPUT PARAMETER replyH AS HANDLE NO-UNDO.
DEFINE VAR rawCust AS RAW.

DO WHILE NOT  DYNAMIC-FUNCTION('endOfStream' IN messageH):
    DYNAMIC-FUNCTION('moveToNext':U IN messageH).
    rawCust = DYNAMIC-FUNCTION('readBytesToRaw':U IN messageH).
    RAW-TRANSFER rawCust TO custt.
    RELEASE custt.
END.

RUN deleteMessage IN messageH.
APPLY "U1" TO THIS-PROCEDURE.

END.
```

3 ♦ Run example14.p.

example14.p

```
/* Publishes the customer table in a Stream message. */
DEFINE VARIABLE pubsubsession AS HANDLE.
DEFINE VARIABLE mesgH AS HANDLE.
DEFINE VARIABLE rawCust AS RAW.

RUN  jms/pubsubsession.p PERSISTENT SET pubsubsession ("-H localhost -S
5162 ").
RUN setBrokerURL IN pubsubsession ("localhost:2506").
RUN beginSession IN pubsubsession.

RUN createStreamMessage IN pubsubsession (OUTPUT mesgH).

FOR EACH customer:
    RAW-TRANSFER customer TO rawCust.
    RUN writeBytesFromRaw IN mesgH(rawCust).
END.

RUN publish IN pubsubsession ("topic1", mesgH, ?, ?, ?).

RUN deleteMessage IN mesgH.
RUN deleteSession IN pubsubsession.
```

Publishing and Receiving a Group Of Messages In a Transaction

Examples 22 and 23 publish and receive a group of messages in a single transaction. Example 22 creates a session that is transacted for sending and Example 23 creates a session that is transacted for receiving.

Follow these steps to publish and receive a group of messages in a transaction:

1 ♦ Run `example23.p` so the subscriber is running before you publish.

example23.p *(1 of 2)*

```
/* Subscribes and receives three messages in a single transaction. */
DEFINE VARIABLE pubsubsession AS HANDLE.
DEFINE VARIABLE consumerH AS HANDLE.
DEFINE VARIABLE msgNum AS INT INIT 0.

/* Creates a transaction for receiving session. */
RUN  jms/pubsubsession.p PERSISTENT SET pubsubsession ("-H localhost -S
5162 ").
RUN setBrokerURL IN pubsubsession ("localhost:2506").
RUN setTransactedReceive IN pubsubsession.
RUN beginSession IN pubsubsession.

/* Subscribe to the TestTopic topic. Messages are handled by the
   "msgHandler" internal procedure.
*/
RUN createMessageConsumer IN pubsubsession (
               THIS-PROCEDURE,   /* This proc will handle it */
               "msgHandler",     /* name of internal procedure */
               OUTPUT consumerH).

RUN subscribe IN pubsubsession (
                    "TestTopic",  /* name of topic */
                    ?,            /* Subscription is not durable */
                    ?,            /* No message selector */
                    no,           /* Want my own messages too */
                    consumerH).   /* Handles the incoming messages*/

/* Start receiving messages */
RUN startReceiveMessages IN pubsubsession.

/* Wait to receive the three messages. */
WAIT-FOR u1 OF THIS-PROCEDURE.
```

example23.p *(2 of 2)*

```
PROCEDURE msgHandler:
DEFINE INPUT PARAMETER messageH AS HANDLE.
DEFINE INPUT PARAMETER msgConsumerH AS HANDLE.
DEFINE OUTPUT PARAMETER replyH AS HANDLE.

/* Display the message - we assume that reply is not required. */
DISPLAY "Message text: " DYNAMIC-FUNCTION('getText':U IN messageH)
  FORMAT "x(70)".
RUN deleteMessage IN messageH.

msgNum = msgNum + 1.

/* Commit the reception of the three messages. */
IF msgNum = 3 THEN
DO:
  RUN commitReceive IN pubsubsession.
  message "committed!".
  APPLY "U1" TO THIS-PROCEDURE.
END.

END.
```

2 ♦ Run example22.p to subscribe and receive messages from `example22.p` in a single transaction.

example22.p

```
/* Publishes A group of Text messages in a single transaction. */
DEFINE VARIABLE pubsubsession AS HANDLE.
DEFINE VARIABLE messageH AS HANDLE.

/* Creates a transcated for sending session. */
RUN  jms/pubsubsession.p PERSISTENT SET pubsubsession ("-H localhost -S
5162 ").
RUN setBrokerURL IN pubsubsession ("localhost:2506").
RUN setTransactedSend IN pubsubsession.
RUN beginSession IN pubsubsession.

/* Create a text message */
RUN createTextMessage IN pubsubsession (OUTPUT messageH).

/* Publish three messages */
RUN setText IN messageH ("message1").
RUN publish IN pubsubsession ("TestTopic", messageH, ?, ?, ?).
RUN setText IN messageH ("message2").
RUN publish IN pubsubsession ("TestTopic", messageH, ?, ?, ?).
RUN setText IN messageH ("message3").
RUN publish IN pubsubsession ("TestTopic", messageH, ?, ?, ?).

/* Commit the publication of the messages. */
RUN commitSend IN pubsubsession.

RUN deleteMessage IN messageH.
RUN deleteSession IN pubsubsession.
```

Installing an Error Handler To Handle an Asynchronous Error

Example 16 installs an error handler to detect a JMS server communication loss.

Follow these steps to install an error handler to handle an asynchronous error:

1 ♦ Run example16.p.

example16.p *(1 of 2)*

```
/* Installs an error handler to deal with a JMS server communication loss.
*/
DEFINE VARIABLE pubsubsession AS HANDLE.
DEFINE VARIABLE errorConsumer AS HANDLE.
DEFINE VARIABLE jmsIsOk AS LOGICAL INIT yes.

RUN  jms/pubsubsession.p PERSISTENT SET pubsubsession ("-H localhost -S
5162 ").
RUN setBrokerURL IN pubsubsession ("localhost:2506").
RUN beginSession IN pubsubsession.
RUN createMessageConsumer IN pubsubsession(THIS-PROCEDURE,
  "errorHandler",
  OUTPUT errorConsumer).
RUN setErrorHandler IN pubsubsession (errorConsumer).
RUN startReceiveMessages IN pubsubsession.

/* Wait forever for messages until the connection with the JMS server is
   lost with error code "-5" (shutdown the SonicMQ broker to simulate
   that).
*/
RUN waitForMessages IN pubsubsession ("inWait", THIS-PROCEDURE, ?).

IF NOT jmsIsOk THEN
DO:
    MESSAGE "Disocnnectiong from JMS Server... "VIEW-AS ALERT-BOX.
    RUN deleteSession IN pubsubsession.
END.
```

```
FUNCTION inWait RETURNS LOGICAL.
RETURN jmsIsOk.
END.
```

example16.p *(2 of 2)*

```
PROCEDURE errorHandler:
DEFINE INPUT PARAMETER messageH AS HANDLE NO-UNDO.
DEFINE INPUT PARAMETER messageConsumerH AS HANDLE NO-UNDO.
DEFINE OUTPUT PARAMETER replyH AS HANDLE NO-UNDO.

DEFINE VAR errorCode AS CHAR NO-UNDO.
DEFINE VAR errorText AS CHAR NO-UNDO.

ASSIGN
  errorCode =  DYNAMIC-FUNCTION
    ('getCharProperty':U IN messageH, "errorCode")
errorText = DYNAMIC-FUNCTION('getText':U IN messageH).
RUN deleteMessage IN messageH.
MESSAGE errorText errorCode VIEW-AS ALERT-BOX.
IF errorCode = "-5" THEN
    jmsIsOk = no.
END.
```

2 ♦ Shut down the SonicMQ broker to simulate the communication loss.

Installing an Error Handler For Synchronous Errors

The procedure in example17.p publishes a TextMessage to a nonexistent topic and handles the error conditions.

example17.p

```
/* Publishes A Text message to an illegal topic name and handles the
   error conditions. */
DEFINE VARIABLE pubsubsession AS HANDLE.
DEFINE VARIABLE messageH AS HANDLE.
DEFINE VARIABLE Successful-publish AS LOGICAL INIT false.

/* Creates a session object. */
RUN  jms/pubsubsession.p PERSISTENT SET pubsubsession ("-H localhost -S 5162
").
RUN setNoErrorDisplay IN pubsubsession (true).
RUN setBrokerURL IN pubsubsession ("localhost:2506").
RUN beginSession IN pubsubsession.

/* Create a text message */
RUN createTextMessage IN pubsubsession (OUTPUT messageH).
RUN setText IN messageH ("Golf shoes on sale today.").

/* Publish the message on the illegal '*' topic */
DO ON ERROR UNDO, LEAVE:
  RUN publish IN pubsubsession ("*", messageH, ?, ?, ?).
  Successful-publish = true.
END.

If NOT Successful-publish THEN
    MESSAGE "Failed to publish to topic '*': " RETURN-VALUE VIEW-AS ALERT-BOX.

RUN deleteMessage IN messageH.
RUN deleteSession IN pubsubsession.
```

14.6.2 PTP Messaging Examples

The PTP examples consist of a sender and a receiver, and each set should run together. Note that queues cannot be generated on the fly by the clients; queues must be created using the administration tool of the SonicMQ broker.

The path for the PTP messaging examples is:

```
<Progress_install_dir>\src\samples\sonicmq\adapter\examples\example*.p
```

Sending a Message To a Queue and Receiving a Message From a Queue

Examples 18 and 19 send and receive a message from a queue.

Follow these steps to send a message to a queue and receive a message from a queue:

1 ♦ Create the GolfQueue queue using the SonicMQ Explorer. (See the *SonicMQ Programming Guide* for information about creating queues.)

2 ♦ Run example18.p to send a TextMessage to the GolfQueue.

example18.p

```
/* Sends A Text message to a queue. */
DEFINE VARIABLE ptpsession AS HANDLE.
DEFINE VARIABLE messageH AS HANDLE.

/* Creates a session object. */
RUN  jms/ptpsession.p PERSISTENT SET ptpsession ("-H localhost -S 5162
").
RUN setBrokerURL IN ptpsession ("localhost:2506").
RUN beginSession IN ptpsession.

/* Create a text message */
RUN createTextMessage IN ptpsession (OUTPUT messageH).
RUN setText IN messageH ("Golf shoes on sale today.").

/* Sends the message to the "GolfQueue" queue */
RUN sendToQueue IN ptpsession ("GolfQueue", messageH, ?, ?, ?).

RUN deleteMessage IN messageH.
RUN deleteSession IN ptpsession.
```

3 ♦ Run example19.p to receive a message from the GolfQueue.

example19.p

```
/* Receives a Text message from a queue. */
DEFINE VARIABLE ptpsession AS HANDLE.
DEFINE VARIABLE consumerH AS HANDLE.

/* Creates a session object. */
RUN  jms/ptpsession.p PERSISTENT SET ptpsession ("-H localhost -S 5162
").
RUN setBrokerURL IN ptpsession ("localhost:2506").
RUN beginSession IN ptpsession.

/*  GolfQueue Messages are handled by the "golfHandler" procedure.  */
RUN createMessageConsumer IN ptpsession (
                  THIS-PROCEDURE,    /* This proc will handle it */
                  "golfHandler",     /* name of internal procedure */
                  OUTPUT consumerH).

RUN receiveFromQueue IN ptpsession (
                   "GolfQueue",   /* name of queue */
                   ?,             /* No message selector */
                   consumerH).    /* Handles incoming messages*/

/* Start receiving messages */
RUN startReceiveMessages IN ptpsession.

/* Wait to receive the messages. Any other IO-blocked statements can be
   used for receiving messages.
*/
WAIT-FOR u1 OF THIS-PROCEDURE.
```

```
PROCEDURE golfHandler:
DEFINE INPUT PARAMETER messageH AS HANDLE.
DEFINE INPUT PARAMETER msgConsumerH AS HANDLE.
DEFINE OUTPUT PARAMETER replyH AS HANDLE.

/* Display the message - we assume that reply is not required. */
DISPLAY "Message text: " DYNAMIC-FUNCTION('getText':U IN messageH)
  FORMAT "x(70)".
RUN deleteMessage IN messageH.

APPLY "U1" TO THIS-PROCEDURE.
END.
```

Achieving Scalable Server Architecture With PTP Queuing

Examples 20 and 21 use PTP queuing to achieve scalable server architecture. Several instances of example20.p send requests to a single JMS queue and receive replies from servers that run example21.p. You can add more instances to handle an increasing volume of requests.

Follow these steps to run Examples 20 and 21:

1 ♦ Create the requestQueue queue using the SonicMQ Explorer.

2 ♦ Run example20.p to send requests to the requestQueue queue.

example20.p *(1 of 2)*

```
/* Sends a request to a queue and receives a reply from the server. */
DEFINE VARIABLE ptpsession AS HANDLE.
DEFINE VARIABLE consumerH AS HANDLE.
DEFINE VARIABLE requestH AS HANDLE.
DEFINE VARIABLE request AS CHAR.

/* Creates a session object. */
RUN  jms/ptpsession.p PERSISTENT SET ptpsession ("-H localhost -S 5162
").
RUN setBrokerURL IN ptpsession ("localhost:2506").
RUN beginSession IN ptpsession.

/* Create a text message */
RUN createTextMessage IN ptpsession (OUTPUT requestH).

/* Creates a consumer for the reply  */
RUN createMessageConsumer IN ptpsession (
                THIS-PROCEDURE,    /* This proc will handle it */
                "replyHandler",    /* name of internal procedure */
                OUTPUT consumerH).

/* Start the reply receiving  */
RUN startReceiveMessages IN ptpsession.

/* Loop forever. */
REPEAT:
  UPDATE request WITH FRAME f1 CENTERED.
  RUN setText IN requestH (request).
  /* Sends a request to the requestQueue and handles the reply in the
     replyHandler internal procedure. */
  RUN requestReply IN ptpsession ( "requestQueue",
                                   requestH,
                                   ?, /* No reply selector. */
                                   consumerH, ?, ?, ?).

  /* Wait for the reply. */
  WAIT-FOR u1 OF THIS-PROCEDURE.
END.
```

example20.p *(2 of 2)*

```
PROCEDURE replyHandler:
DEFINE INPUT PARAMETER replyH AS HANDLE.
DEFINE INPUT PARAMETER msgConsumerH AS HANDLE.
DEFINE OUTPUT PARAMETER responseH AS HANDLE.

/* Display the reply from the server. */
DISPLAY "reply text: " DYNAMIC-FUNCTION('getText':U IN replyH) FORMAT
"X(30)".
RUN deleteMessage IN replyH.

APPLY "U1" TO THIS-PROCEDURE.

END.
```

3 ♦ Run example21.p to receive requests from the requestQueue queue, execute them, and reply to the requester.

example21.p *(1 of 2)*

```
/* This example implements a server who gets requests from a JMS queue,
executes
   the request, and replies to the requester. Run several instances of
this
   server and several instances of a client (example20) to observe the
   scalability of this configuration. */

DEFINE INPUT PARAMETER serverName AS CHAR.

DEFINE VARIABLE ptpsession AS HANDLE.
DEFINE VARIABLE consumerH AS HANDLE.
DEFINE VARIABLE replyMessage AS HANDLE.

/* Creates a session object. */
RUN  jms/ptpsession.p PERSISTENT SET ptpsession ("-H localhost -S 5162
").
RUN setBrokerURL IN ptpsession ("localhost:2506").
RUN beginSession IN ptpsession.

/* Uses one message for all the replies. */
RUN createTextMessage IN ptpsession (OUTPUT replyMessage).

/* receives requests from  the requestQueue */
RUN createMessageConsumer IN ptpsession (
                THIS-PROCEDURE,    /* This proc will handle it */
                "requestHandler",  /* name of internal procedure */
                OUTPUT consumerH).

RUN receiveFromQueue IN ptpsession (
                "requestQueue",   /* request queue */
                ?,                /* No message selector */
                consumerH).       /* Handles the messages */

/* Start receiving requests */
RUN startReceiveMessages IN ptpsession.

/* Process requests forever. */
RUN waitForMessages IN ptpsession ("inWait", THIS-PROCEDURE, ?).
```

example21.p *(2 of 2)*

```
PROCEDURE requestHandler:
DEFINE INPUT PARAMETER requestH AS HANDLE.
DEFINE INPUT PARAMETER msgConsumerH AS HANDLE.
DEFINE OUTPUT PARAMETER replyH AS HANDLE.

DEFINE VAR replyText AS CHAR.

/* Creates a reply message. The reply is sent automatically when control
   returns to the 4GL-To-JMS implementation.
*/
replyText=serverName + " executed " + DYNAMIC-FUNCTION('getText':U IN
requestH).
RUN deleteMessage IN requestH.
replyH = replyMessage.
RUN setText IN replyH (replyText).

END.
```

```
FUNCTION inWait RETURNS LOGICAL.
RETURN true.
END.
```

14.6.3 Multipart Message Example

The following fragment creates a multipart message:

```
/* Create a multipart message */
RUN createMultipartMessage IN ptpsession (OUTPUT messageH).

/* Create a Sonic text message */
RUN createTextMessage IN ptpsession (OUTPUT messagePartH).
RUN setText IN messagePartH (cTextString).

/* Add part to multipart message */
RUN addMessagePart IN messageH (INPUT messagePartH, INPUT contentIDString).

/* Add a memptr part */
RUN addBytesPart IN messageH (INPUT memptr, INPUT contentTypeString,
  INPUT contentIDString).

/* Add a text part */
RUN addTextPart IN messageH (INPUT memptr, INPUT msgTextString,
  INPUT contentTypeString, INPUT contentIDString).
```

First, the fragment creates a multipart message just as it would create any other supported message type. The createMultipartMessage method returns a message handle, which supports methods for adding parts.

Next, the fragment creates a text message and adds it to the multipart message. Each message part has two main identifiers: content type and content ID. Content type identifies the type of part, while content ID identifies a particular part. Since a Sonic text message already has a content type, when the text message is added, only the content ID must be specified.

Finally, the fragment adds a bytes part, comprising an arbitrary set of bytes represented as a MEMPTR. Adding the bytes part resembles adding the text message except that the content type must also be specified.

14.6.4 Gateway Sample Application

The gateway sample application demonstrates a framework for integrating the native 4GL publish and subscribe mechanism (named events) with the 4GL–JMS API. (See the "Integrating With the Native 4GL Publish-and-Subscribe Mechanism" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL.")

Application Files

The sample application manages a set of customer records loaded from the sports.customer table. For each country, there is one instance of the application that manages the subset of customers from that country. The country is specified as an application startup parameter.

The gateway sample application files are in:

```
<Progress_install_dir>\src\samples\sonicmq\adapter\gateway_examples
```

The gateway sample application consists of these three files:

- appDriver.p — Drives the publish and subscribe gateway example
- JMSgateway.p — Establishes a gateway between local and remote publish and subscribe events
- customers.p — Updates customer records from a specified country while keeping the other records identical to the master copy

The main loop of the application is in appDriver.p:

1. The user specifies the Customer.Cust–num value.

2. The application finds the customer and allows the user to update the record if the Customer.Country field matches the startup country.

3. If the Customer.Country field does not match the startup country, the user can only view the customer record.

   Several applications, each managing one country, run concurrently. Each application is connected to a JMS server through a local JMSgateway object. The goal is to keep the records identical across the different locations.

4. When an application modifies a customer record, it publishes the new record through a 4GL PUBLISH CustUpdate call.

   The local JMSgateway object subscribes to the CustUpdate event. It packs the published parameters in a JMS MapMessage and publishes it to the JMS CustUpdate topic.

5. The other JMSgateway objects subscribe to the JMS CustUpdate topic. They receive the JMS MapMessage, unpack the parameters, and publish the updated record locally through a 4GL PUBLISH CustUpdate call.

6. The application picks up the updated record and updates the local copy.

The file, appDriver.p, drives the publish and subscribe gateway example.

appDriver.p

```
/* appDriver.p: Drives the Pub/Sub gateway example. */
DEFINE INPUT PARAM country AS CHAR.

DEFINE VAR customersH AS HANDLE.
DEFINE VAR gatewayH AS HANDLE.
DEFINE VAR custNum AS int.

/* Initialization */
RUN customers.p PERSISTENT SET customersH.
RUN loadCustomers IN customersH.
RUN JMSgateway.p PERSISTENT SET gatewayH ("-H localhost -S 5162 ").

/* Main loop. */
REPEAT:
 custNum = ?.
 UPDATE custNum label "cust-num" WITH FRAME ff CENTERED TITLE "Find Customer".
 RUN updateCustInteractive IN customersH (custNum, country).
END.

RUN deleteGateway IN gatewayH.
```

The file, JMSgateway.p, establishes a gateway between local and remote publish and subscribe events.

JMSgateway.p *(1 of 2)*

```
/* JMSgateway.p: A gateway between local and remote Pub/Sub events. */
DEFINE INPUT PARAM connectionParams AS CHAR.

/* JMS objects */
DEFINE VARIABLE pubsubsession AS HANDLE.
DEFINE VARIABLE outMessage AS HANDLE.
DEFINE VARIABLE consumer AS HANDLE.

/* Raw Transfer Declarations */
DEFINE TEMP-TABLE rawtt FIELD val AS RAW.
CREATE rawtt.
FUNCTION bufferToRaw RETURNS RAW (bufferH AS HANDLE) FORWARD.

/* Initializes the JMS server and subscribes to the CustUpdate topic */
RUN jms/pubsubsession.p PERSISTENT SET pubsubsession (connectionParams).
RUN setBrokerURL IN pubsubsession ("localhost:2506").
RUN beginSession IN pubsubsession.
RUN createMapMessage IN pubsubsession (OUTPUT outMessage).
RUN createMessageConsumer IN pubsubsession (THIS-PROCEDURE,
                                "handleRemoteEvent",
                                OUTPUT consumer).

RUN subscribe IN pubsubsession (
            "CustUpdate", /* Topic name */
            ?,            /* Not durable. */
            ?,            /* No message selector */
            true,         /* No local events please */
            consumer).
RUN startReceiveMessages IN pubsubsession.

/* Subscribes to local CustUpdate events */
SUBSCRIBE TO "CustUpdate" ANYWHERE RUN-PROCEDURE "handleLocalEvent".
```

```
/* Publish locally a remote message from the CustUpdate topic. */
PROCEDURE handleRemoteEvent:
DEFINE INPUT PARAMETER messageH AS HANDLE NO-UNDO.
DEFINE INPUT PARAMETER messageConsumerH AS HANDLE NO-UNDO.
DEFINE OUTPUT PARAMETER replyH AS HANDLE NO-UNDO.

  PUBLISH "CustUpdate"
          (DATE (DYNAMIC-FUNCTION('getChar':U IN messageH, "updateDate")),
          DYNAMIC-FUNCTION('getInt':U IN messageH, "custNum"),
          DYNAMIC-FUNCTION('getBytesToRaw':U IN messageH, "rawCust")).
  RUN deleteMessage IN messageH.
END.
```

JMSgateway.p *(2 of 2)*

```
/* Publish remotely a local CustUpdate event. */
PROCEDURE handleLocalEvent:
DEFINE INPUT PARAMETER dt AS DATE.
DEFINE INPUT PARAMETER custNum AS INT.
DEFINE INPUT PARAMETER custBuffer AS HANDLE.

RUN setString IN outMessage ("updateDate", STRING(dt)).
RUN setInt IN outMessage ("custNum", custNum).
RUN setBytesFromRaw IN outMessage ("rawCust", bufferToRaw(custBuffer)).
RUN publish IN pubsubsession ("CustUpdate", outMessage,  ?,  ?,  ?).
END.
```

```
PROCEDURE deleteGateway:
RUN deleteMessage IN outMessage.
RUN deleteSession IN pubsubsession.
DELETE OBJECT THIS-PROCEDURE.
END.
```

```
FUNCTION bufferToRaw RETURNS RAW (bufferH AS HANDLE):

/* Raw Transfer Variables */
DEFINE VAR rawbuf AS HANDLE.
DEFINE VAR rawCust AS HANDLE.

rawbuf = BUFFER rawtt:HANDLE.
rawCust = rawbuf:buffer-field(1).
bufferH:raw-transfer(true, rawCust).
RETURN rawCust:BUFFER-VALUE.
END.
```

The file, customers.p, updates customer records from a specified country while keeping the other records identical to the master copy.

customers.p

```
/* customers.p: Manages customer records of a specified country and keeps the
other records identical to the master copy. */
DEFINE TEMP-TABLE custt LIKE customer.
DEFINE BUFFER custtUpd FOR custt.
DEFINE VAR custtH AS HANDLE.
DEFINE VAR bufferH AS HANDLE.

/* Getting a handle to a dynamic buffer. */
custtH = TEMP-TABLE custt:HANDLE.
bufferH = custtH:DEFAULT-BUFFER-HANDLE.

/* Subscribes to CustUpdate events. */
SUBSCRIBE TO "CustUpdate" ANYWHERE RUN-PROCEDURE "updateCustFromRaw".
PROCEDURE loadCustomers:
  FOR EACH customer:
      CREATE custt.
      buffer-copy customer to custt.
  END.
END.
```

```
/* Updates a customer from the "correct" country, displays customers from
other countries. */
PROCEDURE updateCustInteractive.
  DEFINE INPUT PARAM custNum AS INT.
  DEFINE INPUT PARAM custCountry AS CHAR.
  FIND custt WHERE custt.cust-num = custNum.

IF custt.country = custCountry THEN
  DO:
    UPDATE custt WITH 2 COL.
    PUBLISH "CustUpdate" (TODAY, custNum, bufferH).
  END.
  ELSE
    DISPLAY custt WITH 2 COL.
END.
```

```
/* Updates a customer record from a RAW value. */
PROCEDURE updateCustFromRaw:
  DEFINE INPUT PARAM dt AS DATE.
  DEFINE INPUT PARAM custNum AS INT.
  DEFINE INPUT PARAM rawCust AS RAW.
FIND custtUpd WHERE custtUpd.cust-num = custNum.
  RAW-TRANSFER rawCust TO custtUpd.
  MESSAGE custNum VIEW-AS ALERT-BOX TITLE "customer updated".
END.
```

Running the Sample Application

Follow these steps to run the sample application:

1 ♦ Start the JMS server and the SonicMQ Adapter.

2 ♦ Start two or more 4GL clients. Each 4GL client calls:

```
RUN appDriver.p (country-name)
```

Each instance should be connected to the Sports database and should start up with a different country.

3 ♦ Update a customer record with one client and watch the others display an ALERT–BOX with the cust–num of the modified customer.

4 ♦ Display the modified customer record at each application instance. All the copies are identical.

A

HLC Library Function Reference

HLC library functions provide an interface between your HLC functions and Progress. This appendix contains the following sections:

- Function Summary
- Function Reference

For more information on using these functions, see Chapter 2, “Host Language Call Interface,” in this book.

A.1 Function Summary

All HLC library function names begin with the pro prefix. The following tables provide a brief description of each function, listed according to their application functionality.

A.1.1 Shared-variable Access

Table A–1 lists HLC library functions that access Progress shared variables. Each function operates on variables with a specific Progress data type.

Table A–1: Functions That Access Progress Shared Variables

Function	Description
`prordc()`	Reads the value of a shared CHARACTER variable.
`prordd()`	Reads the value of a shared DATE variable.
`prordi()`	Reads the value of a shared INTEGER variable.
`prordl()`	Reads the value of a shared LOGICAL variable.
`prordn()`	Reads the value of a shared DECIMAL variable.
`prordr()`	Reads the value of a shared RECID variable.
`prowtc()`	Writes a value to a shared CHARACTER variable.
`prowtd()`	Writes a value to a shared DATE variable.
`prowti()`	Writes a value to a shared INTEGER variable.
`prowtl()`	Writes a value to a shared LOGICAL variable.
`prowtn()`	Writes a value to a shared DECIMAL variable.
`prowtr()`	Writes a value to a shared RECID variable.

A.1.2 Shared-buffer Access

Table A–2 lists HLC library functions that access the fields of a shared buffer defined during a Progress transaction. Each function operates on fields with a specific Progress data type.

Table A–2: Functions That Access Progress Shared Buffers

Function	Description
`profldix()`	Returns the handle to a specified field in a shared buffer.
`prordbc()`	Reads the value of a CHARACTER field in a shared buffer.
`prordbd()`	Reads the value of a DATE field in a shared buffer.
`prordbi()`	Reads the value of an INTEGER field in a shared buffer.
`prordbl()`	Reads the value of a LOGICAL field in a shared buffer.
`prordbn()`	Reads the value of a DECIMAL field in a shared buffer.
`prordbr()`	Reads the value of a RECID field in a shared buffer.
`prowtbc()`	Writes a value to a CHARACTER field in a shared buffer.
`prowtbd()`	Writes a value to a DATE field in a shared buffer.
`prowtbi()`	Writes a value to an INTEGER field in a shared buffer.
`prowtbl()`	Writes a value to a LOGICAL field in a shared buffer.
`prowtbn()`	Writes a value to a DECIMAL field in a shared buffer.
`prowtbr()`	Writes a value to a RECID field in a shared buffer.

A.1.3 Screen Display

Table A–3 lists HLC library functions that enable HLC applications to use the screen display in a manner consistent with Progress display standards.

Table A–3: Functions That Interact With Progress Displays

Function	Description
`proscopn()`	Restores a UNIX terminal to its I/O mode before Progress was run.
`proclear()`	Clears the display.
`prosccls()`	Sets a UNIX terminal to raw mode.
`promsgd()`	Displays a message on the display (in any I/O mode).

A.1.4 Interrupt Handling

Table A–4 lists HLC library functions that allow applications to test for specific Progress interrupts.

Table A–4: Functions That Test For Progress Interrupts

Function	Description
`prockint()`	Tests whether the interrupt key (**STOP** key) has been pressed.

A.1.5 Timer-service Routines

Table A–5 lists HLC library functions that access Progress-managed timer services.

Table A–5: Functions That Access Progress Timer Services

Function	Description
`prosleep()`	Suspends execution for a specified time interval.
`proevt()`	Sets a specified interval timer.
`prowait()`	Suspends execution until a specified interval timer (previously set by proevt()) expires.
`procncel()`	Cancels a specified interval timer previously set by proevt().

A.2 Function Reference

Following is an alphabetical listing of HLC library functions, including calling sequences. Certain library functions accept input parameters that point to character strings containing the names of Progress shared variables (*psymnam*), buffers (*pbufnam*), and fields (*pfldnam*). These names are not case sensitive.

prockint() - Test for Interrupt Key

Tests whether the user pressed the interrupt key (the STOP key or an assigned function key equivalent).

If the user pressed the interrupt key, `prockint()` returns 1; otherwise, `prockint()` returns 0:

SYNTAX

```
int
prockint ( )
```

proclear() - Clear the Display

Clears the display.

On successful completion, `proclear()` returns 0; otherwise, it returns a non-zero value:

SYNTAX

```
int
proclear ( )
```

NOTES

- The `proclear()` function applies only to character-mode systems.
- On character-mode systems, `proclear()` works whether the terminal is in raw or cooked mode. For more information on raw- and cooked-mode terminal I/O, see Chapter 2, "Host Language Call Interface."

SEE ALSO

promsgd() - Display Message, prosccls() - Set Terminal To Raw Mode and Refresh, proscopn() - Set Terminal To Initial Mode

procncel() - Cancel Interval Timer

Cancels a specified interval timer previously set by `proevt()`. You can call this function even if the time interval has already expired:

SYNTAX

```
void
procncel ( pflag )
  char *pflag;
```

pflag

Points to a timer flag previously passed to `proevt()` to start the specified time interval. This flag specifies one of many interval timers that you can set with `proevt()`.

NOTE

- This function applies only to UNIX systems. For more information on using interval timers, see Chapter 2, "Host Language Call Interface."

SEE ALSO

prockint() - Test for Interrupt Key, proevt() - Set Interval Timer, prosleep() - Sleep For Specified Interval, prowait() - Wait For Timer To Expire

proevt() - Set Interval Timer

Sets an interval timer specified by a timer flag. The caller must test the timer flag to know whether the time interval has expired for the corresponding interval timer. The caller must not free the storage containing the flag until either the time interval has expired or `procncel()` has been called to cancel the specified interval timer. You can use `proevt()` to start an unlimited number of times:

SYNTAX

```
void
proevt ( seconds, pflag )
  int  seconds ;
  char *pflag;
```

seconds

Specifies the duration (in seconds) to set the specified interval timer.

pflag

Points to a timer flag used to identify and monitor the specified interval timer. The caller must set **pflag* to 0 before invoking `proevt()`. When the time interval expires, `proevt()` sets **pflag* to a non-zero value. You can test **pflag* explicitly with your own code or implicitly using the `prowait()` function.

NOTE

- This function applies only to UNIX systems. For more information on using HLC timer services, see Chapter 2, “Host Language Call Interface.”

SEE ALSO

procncel() - Cancel Interval Timer, prockint() - Test for Interrupt Key, prosleep() - Sleep For Specified Interval, prowait() - Wait For Timer To Expire

profldix() - Return Field Handle

Returns a field handle for a specified field within a shared buffer. The handle is returned as an integer value. The shared buffer can also refer to a TEMP-TABLE.

On successful completion, `profldix()` returns the field handle for the field; otherwise, it returns a negative value:

SYNTAX

```
int
profldix ( pbufnam, pfldnam )
  char *pbufnam;
  char *pfldnam;
```

pbufnam

Points to the name of the shared buffer containing the specified field. You supply the name from your Progress application.

pfldnam

Points to the name of the specified field within the buffer. You supply the name from your Progress application.

NOTE

- All other shared buffer access functions require the field handle that `profldix()` returns.

SEE ALSO

prordbc() - Read CHARACTER Field, prordbd() - Read DATE Field, prordbi() - Read INTEGER Field, prordbl() - Read LOGICAL Field, prordbn() - Read DECIMAL Field, prordbr() - Read RECID Field, prowtbc() - Write CHARACTER Field, prowtbd() - Write DATE Field, prowtbi() - Write INTEGER Field, prowtbl() - Write LOGICAL Field, prowtbn() - Write DECIMAL Field, prowtbr() - Write RECID Field

promsgd() - Display Message

The `promsgd()` function displays messages on the display.

On successful completion, `promsgd()` returns a 0; otherwise, it returns a non-zero value:

SYNTAX

```
int
promsgd ( pmessage )
  char *pmessage;
```

pmessage

This input parameter points to the message you want to display.

NOTES

- On UNIX character-mode systems, when the terminal is under the control of Progress (in raw-mode terminal I/O), the messages appear at the bottom of the display. If the application previously set the terminal to cooked mode with a call to `proscopn()`, the messages are written to the current output destination. If the current output destination is the display, the messages appear on the display in normal video, similar to the output of the standard C library function `printf()`. For more information on raw- and cooked-mode terminal I/O in character mode, see Chapter 2, “Host Language Call Interface.”
- In Windows, `promsgd()` displays messages in an alert box. Raw and cooked terminal I/O apply only to Progress running on UNIX character-mode systems.

SEE ALSO

proclear() - Clear the Display, prosccls() - Set Terminal To Raw Mode and Refresh, proscopn() - Set Terminal To Initial Mode

prordbc() - Read CHARACTER Field

The `prordbc()` function reads the value of a CHARACTER field in a shared buffer.

On successful completion, `prordbc()` returns 0; otherwise, it returns a non-zero value:

SYNTAX

```
int
prordbc ( pbufnam, fhandle, index, pvar, punknown, varlen,
            pactlen )
  char *pbufnam;
  int   fhandle;
  int   index;
  char *pvar;
  int  *punknown;
  int   varlen;
  int  *pactlen;
```

pbufnam

This input parameter points to the name of the specified shared buffer. You supply the name from your Progress application.

fhandle

This input parameter is the field handle that `profldix()` returns for the specified field.

index

This input parameter specifies an index value for an array field. If the field is not an array, you must set the value of *index* to 0.

pvar

This output parameter points to a buffer where `prordbc()` returns the value of the specified CHARACTER field.

punknown

This output parameter points to an integer where `prordbc()` returns 1 if the field has the UNKNOWN value (?), and returns 0 otherwise.

varlen

This input parameter contains the length of the buffer that *pvar* specifies.

pactlen

This output parameter points to an integer where `prordbc()` returns the actual length (in bytes) of the data in the specified CHARACTER field.

SEE ALSO

profldix() - Return Field Handle, prordbd() - Read DATE Field, prordbi() - Read INTEGER Field, prordbl() - Read LOGICAL Field, prordbn() - Read DECIMAL Field, prordbr() - Read RECID Field, prowtbc() - Write CHARACTER Field

prordbd() - Read DATE Field

The `prordbd()` function reads the value of a DATE field in a shared buffer.

On successful completion, `prordbd()` returns 0; otherwise, it returns a non-zero value:

SYNTAX

```
int
prordbd ( pbufnam, fhandle, index, pyear, pmonth, pday,
          punknown )
  char *pbufnam;
  int   fhandle;
  int   index;
  int  *pyear;
  int  *pmonth;
  int  *pday;
  int  *punknown;
```

pbufnam

This input parameter points to the name of the specified shared buffer.

fhandle

This input parameter is the field handle that `profldix()` returns for the specified field.

index

This input parameter specifies an index value for an array field. If the field is not an array, you must set the value of *index* to 0.

pyear

This output parameter points to an integer where `prordbd()` returns the value of year.

pmonth

This output parameter points to an integer where `prordbd()` returns the value of month.

pday

This output parameter points to an integer where `prordbd()` returns the value of day.

punknown

This output parameter points to an integer where `prordbd()` returns 1 if the field has the UNKNOWN value (?), and returns 0 otherwise.

SEE ALSO

profldix() - Return Field Handle, prordbc() - Read CHARACTER Field, prordbi() - Read INTEGER Field, prordbl() - Read LOGICAL Field, prordbn() - Read DECIMAL Field, prordbr() - Read RECID Field, prowtbd() - Write DATE Field

prordbi() - Read INTEGER Field

The `prordbi()` function reads the value of an INTEGER field in a shared buffer.

On successful completion, `prordbi()` returns 0; otherwise, it returns a non-zero value:

SYNTAX

```
int
prordbi ( pbufnam, fhandle, index, pvar, punknown )
  char *pbufnam;
  int   fhandle;
  int   index;
  long *pvar;
  int  *punknown;
```

pbufnam

This input parameter points to the name of the specified shared buffer. You supply the name from your Progress application.

fhandle

This input parameter is the field handle that `profldix()` returns for the specified field.

index

This input parameter specifies an index value for an array field. If the field is not an array, you must set the value of *index* to 0.

pvar

This output parameter points to a long where `prordbi()` returns the value of the specified INTEGER field.

punknown

This output parameter points to an integer where `prordbi()` returns 1 if the field has the unknown value (?), and returns 0 otherwise.

SEE ALSO

profldix() - Return Field Handle, prordbc() - Read CHARACTER Field, prordbd() - Read DATE Field, prordbl() - Read LOGICAL Field, prordbn() - Read DECIMAL Field, prordbr() - Read RECID Field, prowtbi() - Write INTEGER Field

prordbl() - Read LOGICAL Field

The `prordbl()` function reads the value of a LOGICAL field in a shared buffer.

On successful completion, `prordbl()` returns 0; otherwise, it returns a non-zero value:

SYNTAX

```
int
prordbl ( pbufnam, fhandle, index, pvar, punknown )
  char *pbufnam;
   int    fhandle;
   int    index;
   int   *pvar;
   int   *punknown;
```

pbufnam

This input parameter points to the name of the specified shared buffer. You supply the name from your Progress application.

fhandle

This input parameter is the field handle that `profldix()` returns for the specified field.

index

This input parameter specifies an index value for an array field. If the field is not an array, you must set the value of *index* to 0.

pvar

This output parameter points to an integer where `prordbl()` returns the value of the specified LOGICAL field. The `prordl()` function returns 1 if the LOGICAL field is TRUE, and 0 otherwise.

punknown

This output parameter points to an integer where `prordbl()` returns 1 if the field has the unknown value (?), and returns 0 otherwise.

SEE ALSO

profldix() - Return Field Handle, prordbc() - Read CHARACTER Field, prordbd() - Read DATE Field, prordbi() - Read INTEGER Field, prordbn() - Read DECIMAL Field, prordbr() - Read RECID Field, prowtbl() - Write LOGICAL Field

prordbn() - Read DECIMAL Field

The `prordbn()` function reads the value of a DECIMAL numeric field in a shared buffer.

On successful completion, `prordbn()` returns 0; otherwise, it returns a non-zero value:

SYNTAX

```
int
prordbn ( pbufnam, fhandle, index, pvar, punknown, varlen,
          pactlen )
  char *pbufnam;
   int   fhandle;
   int   index;
  char *pvar;
   int  *punknown;
   int   varlen;
   int  *pactlen;
```

pbufnam

This input parameter points to the name of the specified shared buffer. You supply the name from your Progress application.

fhandle

This input parameter is the field handle that `profldix()` returns for the specified field.

index

This input parameter specifies an index value for an array field. If the field is not an array, you must set the value of *index* to 0.

pvar

This output parameter points to a buffer where `prordbn()` returns the value of the specified DECIMAL numeric field, formatted as a character string.

punknown

This output parameter points to an integer where `prordbn()` returns 1 if the field has the unknown value (?), and returns 0 otherwise.

varlen

This input parameter contains the length of the buffer that *pvar* specifies.

pactlen

This output parameter points to an integer where `prordbn()` returns the actual length (in bytes) of the data in the specified DECIMAL numeric field.

SEE ALSO

profldix() - Return Field Handle, prordbc() - Read CHARACTER Field, prordbd() - Read DATE Field, prordbi() - Read INTEGER Field, prordbl() - Read LOGICAL Field, prordbr() - Read RECID Field, prowtbn() - Write DECIMAL Field

prordbr() - Read RECID Field

The `prordbr()` function reads the value of a RECID field in a shared buffer.

On successful completion, `prordbr()` returns 0; otherwise, it returns a non-zero value:

SYNTAX

```
int
prordbr ( pbufnam, fhandle, index, pvar, punknown )
  char *pbufnam;
   int    fhandle;
   int    index;
   long *pvar;
   int   *punknown;
```

pbufnam

This input parameter points to the name of the specified shared buffer. You supply the name from your Progress application.

fhandle

This input parameter is the field handle that `profldix()` returns for the specified field.

index

This input parameter specifies an index value for an array field. If the field is not an array, you must set the value of *index* to 0.

pvar

This output parameter points to a long where `prordbr()` returns the value of the specified RECID field.

punknown

This output parameter points to an integer where `prordbr()` returns 1 if the field has the unknown value (?), and returns 0 otherwise.

SEE ALSO

profldix() - Return Field Handle, prordbc() - Read CHARACTER Field, prordbd() - Read DATE Field, prordbi() - Read INTEGER Field, prordbl() - Read LOGICAL Field, prordbn() - Read DECIMAL Field, prowtbr() - Write RECID Field

prordc() - Read CHARACTER Variable

The `prordc()` function reads the value of a shared CHARACTER variable.

On successful completion, `prordc()` returns 0; otherwise, it returns a non-zero value:

SYNTAX

```
int
prordc ( psymnam, index, pvar, punknown, varlen, pactlen )
  char *psymnam;
   int    index;
  char *pvar;
   int   *punknown;
   int    varlen;
   int   *pactlen;
```

psymnam

This input parameter points to the name of the specified shared CHARACTER variable. You supply the name from your Progress application.

index

This input parameter specifies an index value for an array variable. If the variable is not an array, you must set the value of *index* to 0.

pvar

This output parameter points to a buffer where `prordc()` returns the value of the specified CHARACTER variable.

punknown

This output parameter points to an integer where `prordc()` returns 1 if the variable has the unknown value (?), and returns 0 otherwise.

varlen

This input parameter contains the length of the buffer that *pvar* specifies.

pactlen

This output parameter points to an integer where `prordc()` returns the actual length (in bytes) of the data in the specified CHARACTER variable.

SEE ALSO

prordd() - Read DATE Variable, prordi() - Read INTEGER Variable, prordl() - Read LOGICAL Variable, prordn() - Read DECIMAL Variable, prordr() - Read RECID Variable, prowtc() - Write CHARACTER Variable

prordd() - Read DATE Variable

The `prordd()` function reads the value of a shared DATE variable.

On successful completion, `prordd()` returns 0; otherwise, it returns a non-zero value:

SYNTAX

```
int
prordd ( psymnam, index, pyear, pmonth, pday, punknown )
  char *psymnam;
   int   index;
   int  *pyear;
   int  *pmonth;
   int  *pday;
   int  *punknown;
```

psymnam

This input parameter points to the name of the specified shared DATE variable. You supply the name from your Progress application.

index

This input parameter specifies an index value for an array variable. If the variable is not an array, you must set the value of *index* to 0.

pyear

This output parameter points to an integer where `prordd()` returns the value of year.

pmonth

This output parameter points to an integer where `prordd()` returns the value of month.

pday

This output parameter points to an integer where `prordd()` returns the value of day.

punknown

This output parameter points to an integer where `prordd()` returns 1 if the variable has the unknown value (?), and returns 0 otherwise.

SEE ALSO

prordc() - Read CHARACTER Variable, prordi() - Read INTEGER Variable, prordl() - Read LOGICAL Variable, prordn() - Read DECIMAL Variable, prordr() - Read RECID Variable, prowtd() - Write DATE Variable

prordi() - Read INTEGER Variable

The `prordi()` function reads the value of a shared INTEGER variable.

On successful completion, `prodi()` returns 0; otherwise, it returns a non-zero value:

SYNTAX

```
int
prordi ( psymnam, index, pvar, punknown )
  char *psymnam;
   int   index;
   long *pvar;
   int  *punknown;
```

psymnam

This input parameter points to the name of the specified shared INTEGER variable. You supply the name from your Progress application.

index

This input parameter specifies an index value for an array variable. If the variable is not an array, you must set the value of *index* to 0.

pvar

This output parameter points to a buffer where `prordi()` returns the value of the specified INTEGER variable.

punknown

This output parameter points to an integer where `prordi()` returns 1 if the variable has the unknown value (?), and returns 0 otherwise.

SEE ALSO

prordc() - Read CHARACTER Variable, prordd() - Read DATE Variable, prordl() - Read LOGICAL Variable, prordn() - Read DECIMAL Variable, prordr() - Read RECID Variable, prowti() - Write INTEGER Variable

prordl() - Read LOGICAL Variable

The `prordl()` function reads the value of a shared LOGICAL variable.

On successful completion, `prordl()` returns 0; otherwise, it returns a non-zero value:

SYNTAX

```
int
prordl ( psymnam, index, pvar, punknown )
  char  *psymnam;
   int    index;
   int   *pvar;
   int   *punknown;
```

psymnam

This input parameter points to the name of the specified shared LOGICAL variable. You supply the name from your Progress application.

index

This input parameter specifies an index value for an array variable. If the variable is not an array, you must set the value of *index* to 0.

pvar

This output parameter points to a buffer where `prordl()` returns the value of the specified LOGICAL variable. The `prordbl()` function returns 1 if the LOGICAL variable is TRUE, and 0 otherwise.

punknown

This output parameter points to an integer where `prordl()` returns 1 if the variable has the unknown value (?), and returns 0 otherwise.

SEE ALSO

prordc() - Read CHARACTER Variable, prordd() - Read DATE Variable, prordi() - Read INTEGER Variable, prordn() - Read DECIMAL Variable, prordr() - Read RECID Variable, prowtl() - Write LOGICAL Variable

prordn() - Read DECIMAL Variable

The `prordn()` function reads the value of a shared DECIMAL numeric variable.

On successful completion, `prordn()` returns 0; otherwise, it returns a non-zero value:

SYNTAX

```
int
prordn ( psymnam, index, pvar, punknown, varlen, pactlen )
  char *psymnam;
   int    index;
  char *pvar;
   int   *punknown;
   int    varlen;
   int   *pactlen;
```

psymnam

This input parameter points to the name of the specified shared DECIMAL numeric variable. You supply the name from your Progress application.

index

This input parameter specifies an index value for an array variable. If the variable is not an array, you must set the value of *index* to 0.

pvar

This output parameter points to a buffer where `prordn()` returns the value of the specified DECIMAL numeric variable, formatted as a character string.

punknown

This output parameter points to an integer where `prordn()` returns 1 if the variable has the unknown value (?), and returns 0 otherwise.

varlen

This input parameter contains the length of the buffer that *pvar* specifies.

pactlen

This output parameter points to an integer where `prordn()` returns the actual length of the character string (in bytes) to which the DECIMAL numeric variable is converted.

SEE ALSO

prordc() - Read CHARACTER Variable, prordd() - Read DATE Variable, prordi() - Read INTEGER Variable, prordl() - Read LOGICAL Variable, prordr() - Read RECID Variable, prowtn() - Write DECIMAL Variable

prordr() - Read RECID Variable

The `prordr()` function reads the value of a shared RECID variable.

On successful completion, `prordr()` returns 0; otherwise, it returns a non-zero value:

SYNTAX

```
int
prordr ( psymnam, index, pvar, punknown )
  char *psymnam;
   int   index;
   long *pvar;
   int  *punknown;
```

psymnam

This input parameter points to the name of the specified shared RECID variable. You supply the name from your Progress application.

index

This input parameter specifies an index value for an array variable. If the variable is not an array, you must set the value of *index* to 0.

pvar

This output parameter points to a long where `prordr()` returns the value of the specified RECID variable.

punknown

This output parameter points to an integer where `prordr()` returns 1 if the variable has the unknown value (?), and returns 0 otherwise.

SEE ALSO

prordc() - Read CHARACTER Variable, prordd() - Read DATE Variable, prordi() - Read INTEGER Variable, prordl() - Read LOGICAL Variable, prordn() - Read DECIMAL Variable, prowtr() - Write RECID Variable

prosccls() - Set Terminal To Raw Mode and Refresh

The `prosccls()` function sets raw mode terminal I/O and refreshes the display. If you do not follow a call to `proscopn()` with a call to `prosccls()`, Progress still calls `prosccls()` to refresh the frames on the display.

On successful completion, `prosccls()` returns 0; if the display is not active (such as in batch mode), it returns 1:

SYNTAX

```
int
prosccls ( restore )
  int restore;
```

restore

If this input parameter has a non-zero value, Progress refreshes the frames on the display. If it is 0, Progress does not refresh the display.

NOTE

- The `prosccls()` function sets raw mode terminal I/O only on character-mode systems. On all other systems, `prosccls()` just refreshes the display. For more information on terminal I/O modes, see Chapter 2, "Host Language Call Interface."

SEE ALSO

proclear() - Clear the Display, promsgd() - Display Message, proscopn() - Set Terminal To Initial Mode

proscopn() - Set Terminal To Initial Mode

The `proscopn()` function sets the terminal to the I/O mode that was in effect when Progress started, which is usually cooked mode.

On successful completion, `proscopn()` returns 0; otherwise, it returns a non-zero value:

SYNTAX

```
int
proscopn ( )
```

NOTE

- The `proscopn()` function applies only to character-mode systems. For more information on terminal I/O modes, see Chapter 2, "Host Language Call Interface."

SEE ALSO

proclear() - Clear the Display, promsgd() - Display Message, prosccls() - Set Terminal To Raw Mode and Refresh

prosleep() - Sleep For Specified Interval

The `prosleep()` function causes your application to suspend execution for a specified time interval. Use `prosleep()` only in special circumstances; signals and interrupts do not interrupt this function:

SYNTAX

```
void
prosleep ( seconds )
   int seconds;
```

seconds

This input parameter specifies the number of seconds to suspend execution.

SEE ALSO

prockint() - Test for Interrupt Key, procncel() - Cancel Interval Timer, proevt() - Set Interval Timer, prowait() - Wait For Timer To Expire

prowait() - Wait For Timer To Expire

The `prowait()` function causes your application to suspend execution until a specified interval timer, set by the `proevt()` function, expires. Use `prowait()` only in special circumstances; signals and interrupts do not interrupt this function:

SYNTAX

```
void
prowait ( pflag )
  char *pflag;
```

pflag

This input parameter points to a timer flag previously passed to `proevt()` in order to start the specified time interval. This flag specifies one of many interval timers you can set with `proevt()`.

NOTE

- This function applies only to UNIX systems. For more information on using interval timers, see Chapter 2, "Host Language Call Interface."

SEE ALSO

prockint() - Test for Interrupt Key, procncel() - Cancel Interval Timer, proevt() - Set Interval Timer, prosleep() - Sleep For Specified Interval

prowtbc() - Write CHARACTER Field

The `prowtbc()` function writes a string value to a CHARACTER field in a shared buffer.

On successful completion, `prowtbc()` returns 0; otherwise, it returns a non-zero value:

SYNTAX

```
int
prowtbc ( pbufnam, fhandle, index, pvar, unknown )
  char *pbufnam;
   int    fhandle;
   int    index;
  char *pvar;
   int    unknown;
```

pbufnam

This input parameter points to the name of the specified shared buffer. You supply the name from your Progress application.

fhandle

This input parameter is the field handle that `profldix()` returns for the specified field.

index

This input parameter specifies an index value for an array field. If the field is not an array, you must set the value of *index* to 0.

pvar

This input parameter points to a buffer containing the character string value for the specified CHARACTER field. The character string must be null-terminated.

unknown

If this input parameter contains the value 1, the unknown value (?) is assigned to the variable. If this input parameter contains the value 0, the input value contained in *pvar* is assigned to the variable.

SEE ALSO

profldix() - Return Field Handle, prordbc() - Read CHARACTER Field, prowtbd() - Write DATE Field, prowtbi() - Write INTEGER Field, prowtbl() - Write LOGICAL Field, prowtbn() - Write DECIMAL Field, prowtbr() - Write RECID Field

prowtbd() - Write DATE Field

The `prowtbd()` function writes a date value to a DATE field in a shared buffer.

On successful completion, `prowtbd()` returns 0; otherwise, it returns a non-zero value:

SYNTAX

```
int
prowtbd ( pbufnam, fhandle, index, year, month, day,
            unknown )
  char *pbufnam;
   int    fhandle;
   int    index;
   int    year;
   int    month;
   int    day;
   int    unknown;
```

pbufnam

This input parameter points to the name of the specified shared buffer. You supply the name from your Progress application.

fhandle

This input parameter is the field handle that `profldix()` returns for the specified field.

index

This input parameter specifies an index value for an array field. If the field is not an array, you must set the value of *index* to 0.

year

This input parameter contains the year value for the specified DATE field.

month

This input parameter contains the month value for the specified DATE field.

day

This input parameter contains the day value for the specified DATE field.

unknown

If this input parameter contains the value 1, the unknown value (?) is assigned to the variable. If this input parameter contains the value 0, the input value contained in *year*, *month*, and *day* is assigned to the variable.

SEE ALSO

profldix() - Return Field Handle, prordbd() - Read DATE Field, prowtbc() - Write CHARACTER Field, prowtbi() - Write INTEGER Field, prowtbl() - Write LOGICAL Field, prowtbn() - Write DECIMAL Field, prowtbr() - Write RECID Field

prowtbi() - Write INTEGER Field

The `prowtbi()` function writes an integer value to an INTEGER field in a shared buffer.

On successful completion, `prowtbi()` returns 0; otherwise, it returns a non-zero value:

SYNTAX

```
int
prowtbi ( pbufnam, fhandle, index, var, unknown )
  char *pbufnam;
   int    fhandle;
   int    index;
   long   var;
   int    unknown;
```

pbufnam

This input parameter points to the name of the specified shared buffer. You supply the name from your Progress application.

fhandle

This input parameter is the field handle that `profldix()` returns for the specified field.

index

This input parameter specifies an index value for an array field. If the field is not an array, you must set the value of *index* to 0.

var

This input parameter contains the integer value for the specified INTEGER field.

unknown

If this input parameter contains the value 1, the unknown value (?) is assigned to the variable. If this input parameter contains the value 0, the input value contained in *var* is assigned to the variable.

SEE ALSO

profldix() - Return Field Handle, prordbi() - Read INTEGER Field, prowtbc() - Write CHARACTER Field, prowtbd() - Write DATE Field, prowtbl() - Write LOGICAL Field, prowtbn() - Write DECIMAL Field, prowtbr() - Write RECID Field

prowtbl() - Write LOGICAL Field

The `prowtbl()` function writes a Boolean value to a LOGICAL field in a shared buffer.

On successful completion, `prowtbl()` returns 0; otherwise, it returns a non-zero value:

SYNTAX

```
int
prowtbl ( pbufnam, fhandle, index, var, unknown )
  char *pbufnam;
   int    fhandle;
   int    index;
   long   var;
   int    unknown;
```

pbufnam

This input parameter points to the name of the specified shared buffer. You supply the name from your Progress application.

fhandle

This input parameter is the field handle that `profldix()` returns for the specified field.

index

This input parameter specifies an index value for an array field. If the field is not an array, you must set the value of *index* to 0.

var

This input parameter contains the boolean value for the specified LOGICAL field. Set *var* to 1 if you want to assign the value of the field to TRUE; set it to 0 otherwise.

unknown

If this input parameter contains the value 1, the unknown value (?) is assigned to the variable. If this input parameter contains the value 0, the input value contained in *var* is assigned to the variable.

SEE ALSO

profldix() - Return Field Handle, prordbl() - Read LOGICAL Field, prowtbc() - Write CHARACTER Field, prowtbd() - Write DATE Field, prowtbi() - Write INTEGER Field, prowtbn() - Write DECIMAL Field, prowtbr() - Write RECID Field

prowtbn() - Write DECIMAL Field

The prowtbn() function writes a decimal value, formatted as a character string, to a DECIMAL numeric field in a shared buffer.

On successful completion, prowtbn() returns 0; otherwise, it returns a non-zero value:

SYNTAX

```
int
prowtbn ( pbufnam, fhandle, index, pvar, unknown )
  char *pbufnam;
   int    fhandle;
   int    index;
  char *pvar;
   int    unknown;
```

pbufnam

This input parameter points to the name of the specified shared buffer. You supply the name from your Progress application.

fhandle

This input parameter is the field handle that profldix() returns for the specified field.

index

This input parameter specifies an index value for an array field. If the field is not an array, you must set the value of *index* to 0.

pvar

This input parameter points to a buffer containing the character string value for the specified DECIMAL numeric field. The character string must be null-terminated.

unknown

If this input parameter contains the value 1, the unknown value (?) is assigned to the variable. If this input parameter contains the value 0, the input value contained in *pvar* is assigned to the variable.

SEE ALSO

profldix() - Return Field Handle, prordbn() - Read DECIMAL Field, prowtbc() - Write CHARACTER Field, prowtbd() - Write DATE Field, prowtbi() - Write INTEGER Field, prowtbl() - Write LOGICAL Field, prowtbr() - Write RECID Field

prowtbr() - Write RECID Field

The `prowtbr()` function writes an integer value to a RECID field in a shared buffer.

On successful completion, `prowtbr()` returns 0; otherwise, it returns a non-zero value:

SYNTAX

```
int
prowtbr ( pbufnam, fhandle, index, var, unknown )
  char *pbufnam;
   int    fhandle;
   int    index;
   long   var;
   int    unknown;
```

pbufnam

This input parameter points to the name of the specified shared buffer. You supply the name from your Progress application.

fhandle

This input parameter is the field handle that `profldix()` returns for the specified field.

index

This input parameter specifies an index value for an array field. If the field is not an array, you must set the value of *index* to 0.

var

This input parameter contains the integer value for the specified RECID field.

unknown

If this input parameter contains the value 1, the unknown value (?) is assigned to the variable. If this input parameter contains the value 0, the input value contained in *var* is assigned to the variable.

SEE ALSO

profldix() - Return Field Handle, prordbr() - Read RECID Field, prowtbc() - Write CHARACTER Field, prowtbd() - Write DATE Field, prowtbi() - Write INTEGER Field, prowtbl() - Write LOGICAL Field, prowtbn() - Write DECIMAL Field

prowtc() - Write CHARACTER Variable

The `prowtc()` function writes a character-string value to a shared CHARACTER variable.

On successful completion, `prowtc()` returns 0; otherwise, it returns a non-zero value.

SYNTAX

```
int
prowtc ( psymnam, index, pvar, unknown )
  char *psymnam;
   int    index;
  char *pvar;
   int    unknown;
```

psymnam

This input parameter points to the name of the specified shared variable. You supply the name from your Progress application.

index

This input parameter specifies an index value for an array variable. If the variable is not an array, you must set the value of *index* to 0.

pvar

This input parameter points to a buffer containing the character string value for the specified CHARACTER variable. The character string must be null-terminated.

unknown

If this input parameter contains the value 1, the unknown value (?) is assigned to the variable. If this input parameter contains the value 0, the input value contained in *pvar* is assigned to the variable.

SEE ALSO

prortc(), prowtd() - Write DATE Variable, prowti() - Write INTEGER Variable, prowtl() - Write LOGICAL Variable, prowtn() - Write DECIMAL Variable, prowtr() - Write RECID Variable

prowtd() - Write DATE Variable

The `prowtd()` function writes a date value to a shared DATE variable.

On successful completion, `prowtd()` returns 0; otherwise, it returns a non-zero value:

SYNTAX

```
int
prowtd ( psymnam, index, year, month, day, unknown )
  char *psymnam;
   int    index;
   int    year;
   int    month;
   int    day;
   int    unknown;
```

psymnam

This input parameter points to the name of the specified shared variable. You supply the name from your Progress application.

index

This input parameter specifies an index value for an array variable. If the variable is not an array, you must set the value of *index* to 0.

year

This input parameter contains the year value for the specified DATE variable.

month

This input parameter contains the month value for the specified DATE variable.

day

This input parameter contains the day value for the specified DATE variable.

unknown

If this input parameter contains the value 1, the unknown value (?) is assigned to the variable. If this input parameter contains the value 0, the input value contained in *year*, *month*, and *day* is assigned to the variable.

SEE ALSO

prortd(), prowtc() - Write CHARACTER Variable, prowti() - Write INTEGER Variable, prowtl() - Write LOGICAL Variable, prowtn() - Write DECIMAL Variable, prowtr() - Write RECID Variable

prowti() - Write INTEGER Variable

The `prowti()` function writes an integer value to a shared INTEGER variable.

On successful completion, `prowti()` returns 0; otherwise, it returns a non-zero value:

SYNTAX

```
int
prowti ( psymnam, index, var, unknown )
  char *psymnam;
   int    index;
   long   var;
   int    unknown;
```

psymnam

This input parameter points to the name of the specified shared variable. You supply the name from your Progress application.

index

This input parameter specifies an index value for an array variable. If the variable is not an array, you must set the value of *index* to 0.

var

This input parameter contains the integer value for the specified INTEGER variable.

unknown

If this input parameter contains the value 1, the unknown value (?) is assigned to the variable. If this input parameter contains the value 0, the input value contained in *var* is assigned to the variable.

SEE ALSO

prorti(), prowtc() - Write CHARACTER Variable, prowtd() - Write DATE Variable, prowtl() - Write LOGICAL Variable, prowtn() - Write DECIMAL Variable, prowtr() - Write RECID Variable

prowtl() - Write LOGICAL Variable

The `prowtl()` function writes a Boolean value to a shared LOGICAL variable.

On successful completion, `prowtl()` returns 0; otherwise, it returns a non-zero value:

SYNTAX

```
int
prowtl ( psymnam, index, var, unknown )
  char *psymnam;
   int    index;
   int    var;
   int    unknown;
```

psymnam

This input parameter points to the name of the specified shared variable. You supply the name from your Progress application.

index

This input parameter specifies an index value for an array variable. If the variable is not an array, you must set the value of *index* to 0.

var

This input parameter contains the boolean value for the specified LOGICAL variable. Set *var* to 1 if you want to assign the value of the variable to TRUE; set it to 0 otherwise.

unknown

If this input parameter contains the value 1, the unknown value (?) is assigned to the variable. If this input parameter contains the value 0, the input value contained in *var* is assigned to the variable.

SEE ALSO

prortl(), prowtc() - Write CHARACTER Variable, prowtd() - Write DATE Variable, prowti() - Write INTEGER Variable, prowtn() - Write DECIMAL Variable, prowtr() - Write RECID Variable

prowtn() - Write DECIMAL Variable

The `prowtn()` function writes a decimal value, formatted as a character string, to a shared DECIMAL numeric variable.

On successful completion, `prowtn()` returns 0; otherwise, it returns a non-zero value:

SYNTAX

```
int
prowtn ( psymnam, index, pvar, unknown )
  char *psymnam;
   int    index;
  char *pvar;
   int    unknown;
```

psymnam

This input parameter points to the name of the specified shared variable. You supply the name from your Progress application.

index

This input parameter specifies an index value for an array variable. If the variable is not an array, you must set the value of *index* to 0.

pvar

This input parameter points to a buffer containing the character string value for the specified DECIMAL numeric variable. The character string must be null terminated.

unknown

If this input parameter contains the value 1, the unknown value (?) is assigned to the variable. If this input parameter contains the value 0, the input value contained in *pvar* is assigned to the variable.

SEE ALSO

prortn(), prowtc() - Write CHARACTER Variable, prowtd() - Write DATE Variable, prowti() - Write INTEGER Variable, prowtl() - Write LOGICAL Variable, prowtr() - Write RECID Variable

prowtr() - Write RECID Variable

The `prowtr()` function writes an integer value to a shared RECID variable.

On successful completion, `prowtr()` returns 0; otherwise, it returns a non-zero value:

SYNTAX

```
int
prowtr ( psymnam, index, var, unknown )
  char *psymnam;
   int    index;
   long   var;
   int    unknown;
```

psymnam

This input parameter points to the name of the specified shared variable. You supply the name from your Progress application.

index

This input parameter specifies an index value for an array variable. If the variable is not an array, you must set the value of *index* to 0.

var

This input parameter contains the integer value for the specified RECID variable.

unknown

If this input parameter contains the value 1, the unknown value (?) is assigned to the variable. If this input parameter contains the value 0, the input value contained in *var* is assigned to the variable.

SEE ALSO

prortr(), prowtc() - Write CHARACTER Variable, prowtd() - Write DATE Variable, prowti() - Write INTEGER Variable, prowtl() - Write LOGICAL Variable, prowtn() - Write DECIMAL Variable

B

COM Object Data Type Mapping

Progress automatically converts data between COM data types and Progress data types for COM object methods and properties, as well as for ActiveX control event parameters. This appendix describes this conversion process and the general support for COM data types in Progress. To perform conversions, Progress uses:

- Options in the 4GL that indicate COM data types
- The Type Libraries installed with each Automation Server or ActiveX control
- The data structures that define the data for each COM object property and parameter

For more information on the syntax for COM object references and Progress support for Type Libraries, see Chapter 7, “Using COM Objects In the 4GL.”

The data conversion support for ActiveX control event parameters is a subset of the general support for COM object data type mapping. For information on this support, see Chapter 9, “ActiveX Control Support.”

This appendix contains the following sections:

B.1 Data Type Conversion Strategy

Progress uses different mechanisms to convert COM data types to Progress data types and to convert Progress data types to COM data types.

B.1.1 Converting COM To Progress Data Types

A COM object passes data to Progress in a Variant structure for COM object properties (get), method output parameters, method return values, and event input parameters. A *Variant* is a self-describing data type whose structure contains both the type of data and the data itself. Progress accesses the information in this structure to convert COM data types to Progress data types.

B.1.2 Converting Progress To COM Data Types

Progress uses 4GL type specifiers and the object's Type Library to convert Progress data types to COM data types for COM object properties (set), method input parameters, and event output parameters. This is the procedure that Progress follows for these conversions:

1. If the 4GL includes type specifiers, Progress performs the conversion based on these specifiers.

2. If the 4GL does not include type specifiers, Progress checks if the COM object provides any type information through its Type Library interfaces. If available, Progress performs the conversion based on that type information.

3. If neither Step 1 nor Step 2 results in a conversion, Progress performs the default conversion.

For information on 4GL type specifiers and how to use them, see Chapter 7, "Using COM Objects In the 4GL."

B.1.3 General Conversion Features and Limitations

If you set a method input parameter, an event output parameter, or a property to the unknown value (?), Progress converts this value to the COM Null value. Similarly, if a COM object returns a Null value in a method output parameter, an event input parameter, or a property, Progress converts this value to the unknown value (?).

Progress does not support conversions between COM Array data types and any Progress data type, including data types with extents (like AS INTEGER EXTENT 5).

When the Progress COM Object Viewer suggests the syntax for a property or method reference or the AppBuilder generates a template for an OCX event procedure, it indicates any method parameter or property with an undetermined data type using the place holder, <anytype>. You must replace <anytype> with a valid Progress data type in your code. For more information on the Progress COM Object Viewer, see Chapter 7, "Using COM Objects In the 4GL." For more information on OCX event procedure templates, see Chapter 9, "ActiveX Control Support."

B.2 Conversions From Progress To COM Data Types

Table B–1 lists the possible conversions from Progress data types to COM data types when setting COM object properties, passing method input parameters, or returning event output parameters.

In addition to the listed conversion pairs, Progress can convert other Progress data types to COM data types when the target COM data type is known and the data is compatible. For example, a character string with the value "123" can be converted to a Currency COM data type, even though Table B–1 does not show a default CHARACTER to Currency conversion pair.

NOTE: The names for COM data types in Table B–1 conform to a nomenclature used in the documentation for most COM objects. For matching alternatives to these names, seen in some documentation and COM object viewers, see Table B–3.

Table B–1: Conversions From Progress To COM Data Types *(1 of 2)*

Progress Data Type	Conversion Information	COM Data Type
CHARACTER	None (default)	String
COM-HANDLE	None (default)	Object (COM)
COM-HANDLE	Type specifier: AS IUNKNOWN or Type Library: Object (Base)	Object (Base)

Table B–1: Conversions From Progress To COM Data Types *(2 of 2)*

Progress Data Type	Conversion Information	COM Data Type
DATE	None (default)	Date
DECIMAL	None (default)	Double
DECIMAL	Type specifier: AS CURRENCY or Type Library: Currency	Currency
DECIMAL	Type specifier: AS FLOAT or Type Library: Single (Float)	Single (Float)
INTEGER	None (default)	Byte
INTEGER	None (default)	Long (4-byte integer)
INTEGER	None (default)	Unsigned Long (4-byte integer)
INTEGER	None (default)	Unsigned Short (2-byte integer)
INTEGER	Type specifier: AS ERROR-CODE or Type Library: Error Code	Error Code
INTEGER	Type specifier: AS SHORT or Type Library: Integer (2-byte integer)	Integer (2-byte integer)
INTEGER	Type specifier: AS UNSIGNED-BYTE or Type Library: Unsigned Byte	Unsigned Byte
LOGICAL	None (default)	Boolean
RAW	None (default)	Byte[]
RAW	None (default)	VT_ARRAY (if single-dimensional array of bytes)
RECID	None (default)	Long (4-byte integer)

B.2.1 Progress To COM Data Type Features and Limitations

When performing Progress to COM data type conversions, Progress does do automatic pointer conversions, but does not do conversions for certain data Progress types.

Pointer Conversions

When indicated, Progress converts the specified Progress data type to a Pointer or Variant Pointer COM data type. This has no effect on the value of the parameter or property, only on the way the value is packaged. Progress determines the conversion in the following ways:

- Converts to a Pointer when the method parameter or property reference includes the BY-POINTER type option, or the Type Library specifies a Pointer to the corresponding COM data type.
- Converts to a Variant Pointer when the method parameter or property reference includes the BY-VARIANT-POINTER type option, or the Type Library specifies a Variant Pointer to the corresponding COM data type.

Progress Data Types Not Converted

Progress does not convert the MEMPTR, ROWID, or WIDGET-HANDLE data types to COM data types.

B.3 Conversions From COM To Progress Data Types

Table B–2 lists the possible conversions from COM data types to Progress data types when accessing COM object properties, method return values, method output parameters, and event input parameters.

If the destination Progress data item has a different data type than the one listed in Table B–2, Progress tries to convert it. Also, if the COM data type returned to Progress does not match any of the COM data types listed in Table B–2, the Progress data item receives the unknown value (?).

NOTE: The names for COM data types in Table B–2 conform to a nomenclature used in the documentation for most COM objects. For matching alternatives to these names, seen in some documentation and COM object viewers, see Table B–3.

Table B–2: Conversions From COM To Progress Data Types *(1 of 2)*

COM data type	Progress Data Type
Boolean	LOGICAL
Byte	INTEGER
Byte[]	RAW
Currency	DECIMAL
Date	DATE
Double	DECIMAL
Integer (2-byte integer)	INTEGER
Long (4-byte integer)	INTEGER
Object (Base)	COM-HANDLE
Object (COM)	COM-HANDLE
Single (Float)	DECIMAL
String	CHARACTER
Unsigned Byte	INTEGER

Table B–2: Conversions From COM To Progress Data Types *(2 of 2)*

COM data type	Progress Data Type
Unsigned Long (4-byte integer)	INTEGER
Unsigned Short (2-byte integer)	INTEGER

NOTE: Progress resolves any Pointer or Variant Pointer references to COM data types, then converts the value to the corresponding Progress data type.

B.4 Alternate COM Data Type Names

Documentation on COM objects generally specifies COM data type names similar to those shown in Table B–1 and Table B–2. However, some documentation and some COM object viewers might use an alternative nomenclature. Table B–3 shows the most common alternates.

Table B–3: Alternative COM Data Type Names *(1 of 2)*

Alternative Name	Common Name
VT_ARRAY	Array
VT_BOOL	Boolean
VT_BSTR	String
VT_BYREF	Pointer
VT_BYREF + VT_VARIANT	Variant Pointer
VT_CY	Currency
VT_DATE	Date
VT_DISPATCH	Object (COM)
VT_ERROR	Error Code
VT_I1	Byte
VT_I2	Integer (2-byte integer)
VT_I4	Long (4-byte integer)

Table B–3: Alternative COM Data Type Names *(2 of 2)*

Alternative Name	Common Name
VT_PTR	Pointer
VT_PTR + VT_VARIANT	Variant Pointer
VT_R4	Single (Float)
VT_R8	Double
VT_UI1	Unsigned Byte
VT_UI2	Unsigned Short (2-byte integer)
VT_UI4	Unsigned Long (4-byte integer)
VT_UNKNOWN	Object (Base)
VT_VARIANT	Variant (<anytype>)

C

4GL–JMS API Reference

This appendix provides reference information on the procedures and functions (and a connection parameter and a global variable) for the Sonic MQ Adapter 4GL–JMS API.

This information is presented in the following sections:

C.1 4GL–JMS API Cross-reference

The following tables list the methods in the:

- Session Objects
- Message Consumer objects
- Message 0bjects

C.1.1 Methods In the Session Objects

Table C–1 provides a **brief** description of the methods in the Session Objects. For complete information and syntax, see the "Session Objects" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL" or the "4GL–JMS API Alphabetical Reference" section in this appendix.

Table C–1: Methods In the Session Objects *(1 of 5)*

Method	Purpose
beginSession	Procedure connects to the SonicMQ Adapter and starts a JMS connection and session.
browseQueue	Procedure allows applications to view messages in a queue without consuming them.
cancelDurableSubscription	Procedure cancels a durable subscription.
commitReceive	Procedure acknowledges all messages received up to that point in the current transaction.
commitSend	Procedure sends all messages published (or sent to a queue) up to that point in the current transaction.
createBytesMessage	Procedure creates a new BytesMessage.
createHeaderMessage	Procedure creates a new HeaderMessage.
createMapMessage	Procedure creates a new MapMessage.
createMessageConsumer	Procedure creates a new Message Consumer.
createMultipartMessage	Procedure creates a multipart message and returns a handle to it.

Table C–1: Methods In the Session Objects *(2 of 5)*

Method	Purpose
createStreamMessage	Procedure creates a new StreamMessage.
createTextMessage	Procedure creates a new TextMessage.
createXMLMessage	Procedure creates a new XMLMessage.
deleteSession	Procedure closes a session and its underlying connection and deletes the session procedure.
getAdapterService	Function returns the value set by the preceding setAdapterService. Null is returned if setAdapterService was not called.
getBrokerURL	Function returns the value set by the preceding setBrokerURL.
getClientID	Function returns the value set by the preceding setClientID. Null is returned if setClientID was not called.
getConnectionID	Function returns the AppServer connection ID.
getConnectionMetaData	Function returns a comma-separated list of connection and provider attributes.
getConnectionURLs	Function returns comma-separated list of Sonic Broker URLs for the client to try to connect to.
getDefaultPersistency	Function returns the value specified by setDefaultPersistency value.
getDefaultPriority	Function returns the value specified by setDefaultPriority; it returns 4 if setDefaultPriority was not called.
getDefaultTimeToLive	Function returns the value specified by setDefaultTimeToLive.
getLoadBalancing	Function returns the value specified by setLoadBalancing.
getJMSServerName	Function returns the value set by the preceding setJmsServerName.

Table C–1: Methods In the Session Objects *(3 of 5)*

Method	Purpose
getPassword	Function returns the value set by the preceding setPassword.
getSequential	Function returns the value set by the preceding setSequential.
getSingleMessageAcknowledgement	Function returns the value set by the preceding setSingleMessageAcknowledgement
getTransactedReceive	Function returns the value set by the preceding setTransactedReceive.
getTransactedSend	Function returns the value set by the preceding setTransactedSend.
getUser	Function returns the value set by the preceding setUser.
publish	Procedure publishes a message to topicName.
receiveFromQueue	Procedure receives messages from queueName.
recover	Procedure starts redelivering all unacknowledged messages received up to that point in the current session.
requestReply	Procedure sends a message to a destination and designates the messageConsumer parameter for processing replies.
rollbackReceive	Procedure starts redelivering the messages received up to that point in the current transaction.
rollbackSend	Procedure discards all messages sent up to that point in the current transaction.
sendToQueue	Procedure sends a message to queueName.
setAdapterService	Procedure specifies the service name under which the SonicMQ Adapter is registered with the NameServer; the default is `adapter.progress.jms`.

Table C–1: Methods In the Session Objects *(4 of 5)*

Method	Purpose
setBrokerURL	Procedure sets the value of the SonicMQ broker URL.
setClientID	Procedure sets the clientID value for the SonicMQ broker connection and overwrites the default clientID set on the server side.
setConnectionURLs	Procedure specifies a list of brokerURLs for the client to try to connect to.
setDefaultPersistency	Procedure sets the default message persistency value for all messages sent in that session.
setDefaultPriority	Procedure sets the default message priority for all messages sent in that session.
setDefaultTimeToLive	Procedure sets the default time to live.
setErrorHandler	Procedure handles asynchronous conditions.
setJMSServerName	Procedure specifies the JMS broker implementation, SonicMQ.
setLoadBalancing	Procedure enables or disables client-side load balancing.
setNoErrorDisplay	Procedure turns the automatic display of synchronous errors and conditions on and off.
setPassword	Procedure sets the password value for the SonicMQ broker login and overwrites the default password property set on the SonicMQ Adapter side.
setPingInterval	Procedure specifies the interval in seconds for the JMS Adapter to actively ping the SonicMQ broker so communication failure can be detected promptly.
setPrefetchCount	Procedure sets the number of messages a Sonic client can retrieve in a single operation from a queue containing multiple messages.

Table C–1: Methods In the Session Objects *(5 of 5)*

Method	Purpose
setPrefetchThreshold	Procedure determines when the Sonic client goes back to the broker for more messages.
setSequential	Procedure lets the application control whether clients try connecting to brokers in a connection list sequentially or randomly. To attempt load balancing, request randomly.
setSingleMessageAcknowledgement	Procedure lets an application turn on single-message acknowledgement for a client session.
setTransactedReceive	Procedure makes a session transacted for receiving; a session is not transacted by default.
setTransactedSend	Procedure makes a session transacted for sending; a session is not transacted by default.
setUser	Procedure sets the user value for the SonicMQ broker login and overwrites the default user property set on the SonicMQ Adapter side.
startReceiveMessages	Procedure starts receiving messages after creating a new session or after calling stopReceiveMessages.
stopReceiveMessages	Procedure causes the SonicMQ Adapter broker to stop receiving messages on behalf of the 4GL client and to stop sending messages already received by the SonicMQ Adapter broker for the 4GL client.
subscribe	Procedure subscribes to topicName. The messages are handled asynchronously by the messageConsumer object.
waitForMessages	Procedure waits and processes events as long as the user-defined function returns true and there is no period of more than the specified number of seconds in which no messages are received.

C.1.2 Methods In the Message Consumer Object

Table C–2 provides a **brief** description of the methods in the Message Consumer object. For complete information and syntax, see the "Message Consumer Objects" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL"or the "4GL–JMS API Alphabetical Reference" section in this appendix.

Table C–2: Methods In the Message Consumer Object *(1 of 2)*

Function or Procedure	Purpose
acknowledgeAndForward	Procedure causes a message to be forwarded and acknowledged in a single, atomic operation.
deleteConsumer	Procedure ends the life of a Message Consumer.
getApplicationContext	Function returns application context information.
getDestinationName	Function returns the name of the destination that messages arrive from when the message consumer was passed to subscribe or receiveFromQueue.
getNoAcknowledge	Function returns true if setNoAcknowledge was called.
getProcHandle	Function returns the handle to a procedure that contains the name of an internal procedure for handling messages.
getProcName	Function returns the name of the internal procedure for handling messages.
getReplyAutoDelete	Function returns the values set by setReplyAutoDelete.
getReplyPersistency	Function returns the setReplyPersistency value; it is PERSISTENT if setReplyPersistency was not called.
getReplyPriority	Function returns the setReplyPriority value; it is 4 if setReplyPriority was not called.
getReplyTimeToLive	Function returns the setReplyTimeToLive value; it is UNKNOWN if setReplyTimeToLive was not called.
getReuseMessage	Function returns true if setReuseMessage was called; if not, it returns false.
getSession	Function returns a handle to the session.

Table C–2: Methods In the Message Consumer Object *(2 of 2)*

Function or Procedure	Purpose
inErrorHandling	Function returns true when called from a message handler if the application is handling an error message.
inMessageHandling	Function returns true when called from a message handler if the application is handling the data in a subscription (or queue) message.
inQueueBrowsing	Function returns true when called from a message handler if an application is handling a queue browsing message.
inReplyHandling	Function returns true when called from a message handler if an application is handling a reply message.
setApplicationContext	Procedure passes application context to the message handler.
setNoAcknowledge	Procedure instructs the 4GL–JMS implementation not to acknowledge this message.
setReplyPersistency	Procedure sets the value for message persistency when the message consumer is passed to requestReply.
setReplyPriority	Procedure sets the priority of the reply messages when the message consumer is passed to requestReply.
setReplyTimeToLive	Procedure sets the time to live value of the reply messages when the message consumer is passed to requestReply.
setReuseMessage	Procedure instructs the Message Consumer object not to create a new message for each received message.

C.1.3 Methods In the Message Objects

Table C–3 provides a **brief** description of the methods in the Message Objects. For complete information and syntax, see the "Message Objects" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL"or the "4GL–JMS API Alphabetical Reference" section in this appendix.

Table C–3: Methods In the Message Objects *(1 of 8)*

Function or Procedure	Purpose
addBytesPart	Procedure adds any arbitrary part to a multipart message.
addMessagePart	Procedure adds a Sonic-message part to a multipart message.
addTextPart	Procedure adds a text part to a multipart message.
appendText	Procedure, called in write-only mode, to append text to the message in several calls to overcome the Progress 32K limit on the number of characters.
clearBody	Procedure clears the body of a message, keeping header and property values unchanged and transfers a StreamMessage, TextMessage, XMLMessage, and BytesMessage to write-only mode.
clearProperties	Procedure clears the properties of the message, keeping header and body values unchanged.
createMultipartMessage	Procedure creates a multipart message and returns a handle to it.
deleteMessage	Procedure deletes a message.
endOfStream	Function returns true if an application retrieved the last item of a stream, the last bytes segment, or last text segment.
getBytesCount	Function returns the number of bytes in a BytesMessage.
getBytesPartByID	Function converts from a JMS data type and gets a bytes item from a MapMessage.
getBytesPartByIndex	Function retrieves a binary part corresponding to the index and returns the content type as a MEMPTR

Table C–3: Methods In the Message Objects *(2 of 8)*

Function or Procedure	Purpose
getBytesToRaw	Function gets a bytes item from a MapMessage.
getChar	Function returns an item of any data type except bytes from a MapMessage.
getCharCount	Function returns the total number of characters in a message.
getCharProperty	Function returns message properties of any data type.
getConnectionURLs	Function returns any numeric item from a MapMessage.
getDecimal	Function gets any numeric item from a MapMessage.
getDecimalProperty	Function returns any numeric message property.
getInt	Function returns int, short, or bytes items from a MapMessage.
getIntProperty	Function returns int, short, or bytes message properties.
getItemType	Function returns the data type of an item in a MapMessage. UNKNOWN is returned if the item does not exist.
getJMSCorrelationID	Function returns the correlationID. This value is application-defined, typically the ID of the message replied to.
getJMSCorrelationIDAsBytes	Function returns a proprietary (JMS provider-dependent) correlation ID.
getJMSDeliveryMode	Function returns the delivery mode.
getJMSDestination	Function returns the name of the destination this message was sent to.
getJMSExpiration	Function returns the expiration time (GMT).
getJMSMessageID	Function returns the message ID, a unique ID that the JMS server assigns to each message.
getJMSPriority	Function returns priority values in the range of 0–9.

Table C–3: Methods In the Message Objects *(3 of 8)*

Function or Procedure	Purpose
getJMSRedelivered	Function returns true (at the receiver side) if this is not the first delivery of this message.
getJMSReplyTo	Function returns the reply destination.
getJMSTimestamp	Function returns the message sending time, which is the difference in milliseconds between the message creation time and midnight, January 1, 1970 UTC.
getJMSType	Function returns a proprietary (JMS provider-dependent) type name.
getLoadBalancing	Function returns a LOGICAL value indicating whether load balancing is enabled — that is, whether the client is willing to have a connect request redirected to another broker within a SonicMQ cluster.
getLogical	Function returns a BOOLEAN item by name from a MapMessage.
getLogicalProperty	Function returns a boolean message property.
getMapNames	Function returns a comma-separated list of the item names in a MapMessage.
getMEMPTR	Function returns a reference to a MEMPTR variable that contains exactly all the bytes of a BytesMessage and implicitly calls reset.
getMessagePartByID	Function returns the SonicMQ Adapter message type: TextMessage, MapMessage, StreamMessage, BytesMessage, HeaderMessage, or XMLMessage.
getMessagePartByIndex	Function retrieves a handle to the message part corresponding to the index and returns the content type as a character string.
getMessageType	Function returns the SonicMQ Adapter message type: TextMessage, MapMessage, StreamMessage, BytesMessage, HeaderMessage, XMLMessage, or multipartMessage.

Table C–3: Methods In the Message Objects *(4 of 8)*

Function or Procedure	Purpose
getPartCount	Function returns the number of parts in a multipart message.
getPropertyNames	Function returns a comma-separated list of the properties of a message.
getPropertyType	Function returns the message property's data type. UNKNOWN is returned if the property was not set in the message.
getReplyToDestinationType	Function returns "queue," "topic," or UNKNOWN.
getSequential	Function returns a LOGICAL value indicating how a fail-over list is used — that is, whether clients try to connect to brokers in a connection list sequentially or randomly.
getText	Function returns all the text in the message and then implicitly calls reset.
getTextPartByID	Function can be called in read-only mode to get the next text segment when handling large messages.
getTextPartByIndex	Function retrieves the text part corresponding to the index and returns the content type as a CHARACTER string.
getTextSegment	Function can be called in read-only mode to return the next text segment when handling large messages.function can be called in read-only mode to return the next text segment when handling large messagesfunction can be called in read-only mode to return the next text segment when handling large messagesfunction can be called in read-only mode to return the next text segment when handling large messages
hasReplyTo	Function returns true if the JMSreplyTo header was set.
isMessagePart	Function returns TRUE if the part corresponding to a specified index is a SonicMQ message.
messageHandler	This procedure handles incoming JMS and error messages.

Table C–3: Methods In the Message Objects *(5 of 8)*

Function or Procedure	Purpose
moveToNext	Function moves the cursor to the next data item and returns its data type: UNKNOWN, boolean, byte, short, char, int, long, float, double, string, or bytes.
readBytesToRaw	Function returns bytes data from the body of a StreamMessage. It can be called in read-only mode to return the next bytes segment in a BytesMessage.
readChar	Function returns any message data except bytes data from the body of a StreamMessage.
readDecimal	Function returns any numeric data from the body of a StreamMessage.
readInt	Function returns int, short, and bytes data from the body of a StreamMessage.
readLogical	Function returns boolean data from the body of a StreamMessage.
reset	Procedure changes the mode of a message from write-only to read-only mode and positions the cursor before the first segment.
setBoolean	Procedure converts data in a MapMessage to the JMS boolean data type.
setBooleanProperty	Procedure sets a boolean message property. An UNKNOWN value is considered a false value.
setByte	Procedure converts data in a MapMessage to the JMS bytes data type; byte values are -128 to 127. The server returns a NumberFormatException message for a value overflow.
setByteProperty	Procedure sets a bytes property in a message; the values range from –128 to 127. The server returns a NumberFormatException message for a value overflow.
setBytesFromRaw	This procedure converts data in a MapMessage to the JMS bytes data type.

Table C–3: Methods In the Message Objects *(6 of 8)*

Function or Procedure	Purpose
setChar	Procedure converts data in a MapMessage to the JMS char data type; the number of characters in the char value must be one.
setDouble	Procedure converts data in a MapMessage to the JMS double data type.
setDoubleProperty	Procedure sets a double property in a message.
setFloat	Procedure converts data in a MapMessage to the JMS float data type.
setFloatProperty	Procedure sets a float property in a message.
setInt	Procedure converts data in a MapMessage to the JMS int data type.
setIntProperty	Procedure sets a JMS int property in a message.
setJMSCorrelationID	Procedure sets the correlationID. This value is application-defined; typically the ID of the message replied to.
setJMSCorrelationIDAsBytes	Procedure sets the bytes correlationID, a proprietary (JMS provider-dependent) value. When accessing SonicMQ, the bytesCorrelationID field can be used for storing application-defined values.
setJMSReplyTo	Procedure sets a destination for replies.
setJMSType	Procedure sets the type name, which is proprietary (JMS provider-dependent).
setLong	Procedure sets long data; any fractional part of the DECIMAL value is truncated.
setLongProperty	Procedure sets a long message property; any fractional part of the DECIMAL value is truncated.
setMEMPTR	Procedure sets the specified number of bytes from the MEMPTR variable starting at startIndex (the first byte is 1) in a BytesMessage.

Table C–3: Methods In the Message Objects *(7 of 8)*

Function or Procedure	Purpose
setNoErrorDisplay	Procedure turns the automatic display of synchronous errors and conditions on and off.
setReplyToDestinationType	Procedure sets the type of the destination specified by setJMSReplyTo; the type can be queue or topic.
setSequential	Procedure converts data in a MapMessage to the JMS short data type. The server returns a NumberFormatException message for a value overflow.
setShort	Procedure converts data to the JMS short data type in a MapMessage.
setShortProperty	Procedure sets a short message property. The server returns a NumberFormatException message for a value overflow.
setSingleMessageAcknowledgement	Procedure converts data in a MapMessage to the JMS String data type.
setString	Procedure converts data in a MapMessage to the JMS String data type.
setStringProperty	Procedure sets a String property in a message.
setText	Procedure clears the message body and sets a new text value.
writeBoolean	Procedure writes boolean data to the body of a StreamMessage. An UNKNOWN value is considered false.
writeByte	Procedure writes bytes data to the body of a StreamMessage; byte values are –128 to 127. The server returns a NumberFormatException message for a value overflow.
writeBytesFromRaw	Procedure writes bytes data to the body of a StreamMessage. Procedure can be called in write-only mode to write additional bytes segments to a BytesMessage and work around the RAW data type limit of 32K.

Table C–3: Methods In the Message Objects *(8 of 8)*

Function or Procedure	Purpose
writeChar	Procedure writes JMS char data to a message; the number of characters in the char value must be one.
writeDouble	Procedure writes double data to a StreamMessage.
writeFloat	Procedure writes float data to a StreamMessage.
writeInt	Procedure writes JMS int data to a StreamMessage.
writeLong	Procedure writes long data to a StreamMessage. The fractional part of the DECIMAL value is truncated.
writeShort	Procedure writes short data to a StreamMessage. The server returns a NumberFormatException message for a value overflow.
writeString	Procedure writes String data to a StreamMessage.

C.2 4GL–JMS API Alphabetical Reference

This section describes the procedures and functions (and a connection parameter and a global variable) that you can use with the 4GL–JMS API. For this information in context, see the "Programming With the 4GL–JMS API" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

acknowledgeAndForward

In Message Consumer Objects

This procedure applies inside a message event handler. The session must be set to SINGLE_MESSAGE_ACKNOWLEDGE.

The procedure expects a destination queue name, the original message handle, and optional message-delivery properties. If the message-delivery properties are set to ? (the unknown value), the procedure uses the original values from the message.

If the procedure is successful, the message is forwarded and acknowledged in a single, atomic operation. If the procedure is not successful — for example, if the destination does not exist — the message is not acknowledged and eventually returns to the queue:

SYNTAX

```
PROCEDURE acknowledgeAndForward.
DEFINE INPUT PARAMETER destinationName AS CHARACTER.
DEFINE INPUT PARAMETER messageH        AS HANDLE.
DEFINE INPUT PARAMETER priority        AS INTEGER.
DEFINE INPUT PARAMETER timeToLive      AS DECIMAL.
DEFINE INPUT PARAMETER persistency     AS CHARACTER.
```

For information on this procedure, see the "Acknowledge and Forward" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

adapterConnection

This input parameter specifies the connection parameters to the SonicMQ Adapter and allows an application to set different session level attributes before starting the JMS session. An application creates a session procedure by calling `pubsubsession.p` or `ptpsession.p` persistently with the adapterConnection input parameter:

SYNTAX

```
DEFINE INPUT PARAMETER adapterConnection AS CHAR.
```

Examples of valid adapterConnection parameters include:

```
"-H host1 -S 5162" /*The Name Server is on host1, listening on UDP port 5162*/
""            /*The Name Server is local, listening on the default UDP port*/
“-URL http://host1:3099/uri?AppService=adapter.progress.jms”
              /*Connecting through HTTP (on a WebClient)*/
“-URL AppServer://host1:5162/uri?AppService=adapter.progress.jms”
              /*Equivalent to "-H host1 -S 5162"*/
```

For information on the adapterConnection parameter in context, see the “Creating a Session Procedure” section in Chapter 13, “Accessing SonicMQ Messaging From the Progress 4GL.”

For information on `pubsubsession.p` and `ptpsession.p`, see the “Session Objects,” “Creating a Session Procedure,”“pubsubsession.p,” and “ptpsession.p” sections in Chapter 13, “Accessing SonicMQ Messaging From the Progress 4GL.”

addBytesPart

In Message Objects

This procedure adds any arbitrary part to a multipart message. The part can be text or binary. The Sonic message is created as usual.

A content type and a content ID must be specified.

NOTE: To conserve resources, after calling this procedure, the application must delete the MEMPTR (represented by `memptr`):

SYNTAX

```
PROCEDURE addBytesPart.
DEFINE INPUT PARAMETER memptr AS MEMPTR.
DEFINE INPUT PARAMETER contentTypeString AS CHARACTER.
DEFINE INPUT PARAMETER contentIDString AS CHARACTER.
```

For information on this procedure in context, see the "MultipartMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

addMessagePart

In Message Objects

This procedure adds a Sonic message to a multipart message. The Sonic message is created as usual. Its content type is defined by Sonic. The content-ID string (represented by `contentIDString`) sets the content ID of the part and is used to identify it

NOTE: To conserve resources, after calling this procedure, the application must delete the message-part handle (represented by `messagePartH`):

SYNTAX

```
PROCEDURE addMessagePart.
DEFINE INPUT PARAMETER messagePartH AS HANDLE.
DEFINE INPUT PARAMETER contentIDString AS CHARACTER.
```

For information on this procedure, see the "MultipartMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

addTextPart

In Message Objects

This procedure adds a text part to a multipart message.

The method resembles the addBytesPart method except that it takes a CHARACTER string instead of a MEMPTR:

SYNTAX

```
PROCEDURE addTextPart.
DEFINE INPUT PARAMETER charString        AS CHARACTER.
DEFINE INPUT PARAMETER contentTypeString AS CHARACTER.
DEFINE INPUT PARAMETER contentIDString   AS CHARACTER.
```

For information on this procedure, see the "MultipartMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

appendText

In Message Objects

This procedure can be called in write-only mode to append text to the message in several calls to overcome the Progress 32K limit on the number of characters:

SYNTAX

```
PROCEDURE appendText.
DEFINE INPUT PARAMETER textValue AS CHAR.
```

For information on this procedure in context, see the "Setting Text" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

beginSession

In Session Objects

This procedure connects to the SonicMQ Adapter and starts a JMS connection and session. If beginSession returns an error, the Session object is automatically deleted:

SYNTAX

```
PROCEDURE beginSession.
```

NOTE: This procedure executes remotely (sends a message to the SonicMQ Adapter).

For information on this procedure in context, see the "Connecting To the SonicMQ Adapter" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

browseQueue

In Session Objects

This procedure allows applications to view messages in a queue without consuming them. This procedure receives (for browsing) all messages currently in the queue in the messageConsumer object:

SYNTAX

```
PROCEDURE browseQueue.
DEFINE INPUT PARAMETER queueName AS CHAR.
DEFINE INPUT PARAMETER messageConsumer AS HANDLE.
```

NOTES:

- Browsed messages are not removed from the queue or acknowledged and are not subject to the transactional context of the session. (For more information, see the *Java Message Service* specification and the *SonicMQ Programming Guide* on queue browsing).
- This procedure executes remotely (sends a message to the SonicMQ Adapter).

For information on this procedure in context, see the "Browsing Messages On a Queue" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

cancelDurableSubscription

In Session Objects

This procedure cancels a durable subscription. It is an error to call this procedure if there is an active message consumer for that subscription. Call deleteConsumer first to delete the message consumer:

SYNTAX

```
PROCEDURE cancelDurableSubscription.
DEFINE INPUT PARAMETER subscriptionName AS CHAR.
```

NOTE: This procedure executes remotely (sends a message to the SonicMQ Adapter).

For information on this procedure in context, see the "Subscribing To a Topic" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

clearBody

In Message Objects

This procedure clears the body of a message, keeping header and property values unchanged. This procedure transfers a StreamMessage, TextMessage, XMLMessage, and BytesMessage to write-only mode:

SYNTAX

```
PROCEDURE clearBody.
```

For information on this procedure in context, see the "Setting Message Properties" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

clearProperties

In Message Objects

This procedure clears the properties of the message, keeping header and body values unchanged:

SYNTAX

```
PROCEDURE clearProperties.
```

For information on this procedure in context, see the "Setting Message Properties" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

commitReceive

In Session Objects

This procedure acknowledges all messages received up to that point in the current transaction. It is an error to call this method in a Session object that is not transacted for receiving:

SYNTAX

```
PROCEDURE commitReceive.
```

NOTE: This procedure executes remotely (sends a message to the SonicMQ Adapter).

For information on this procedure in context, see the "Transaction and Recovery Methods" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

commitSend

In Session Objects

This procedure sends all messages published (or sent to a queue) up to that point in the current transaction. It is an error to call this method in a Session object that is not transacted for sending:

SYNTAX

```
PROCEDURE commitSend.
```

NOTE: This procedure executes remotely (sends a message to the SonicMQ Adapter).

For information on this procedure in context, see the "Transaction and Recovery Methods" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

createBytesMessage

In Session Objects

This procedure creates a new BytesMessage:

SYNTAX

```
PROCEDURE createBytesMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.
```

For information on this procedure in context, see the "Factory Procedures" and "BytesMessage" sections in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

createHeaderMessage

In Session Objects

This procedure creates a new HeaderMessage:

SYNTAX

```
PROCEDURE createHeaderMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.
```

For information on this procedure in context, see the "Factory Procedures" and "Header Messages" sections in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

createMapMessage

In Session Objects

This procedure creates a new MapMessage:

SYNTAX

```
PROCEDURE createMapMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.
```

For information on this procedure in context, see the "Factory Procedures" and "MapMessage" sections in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

createMessageConsumer

In Session Objects

This procedure creates a new Message Consumer object. The application must pass to createMessageConsumer, the name of an internal procedure for handling messages and a handle to a procedure that contains procName:

SYNTAX

```
PROCEDURE createMessageConsumer.
DEFINE INPUT PARAMETER procHandle AS HANDLE.
DEFINE INPUT PARAMETER procName AS CHAR.
DEFINE OUTPUT PARAMETER consumerHandle AS HANDLE.
```

For information on this procedure in context, see the "Factory Procedures" and the "Message Consumer Objects" sections in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

createMultipartMessage

In Session Objects

This procedure creates a multipart message and returns a handle to it:

SYNTAX

```
PROCEDURE createMultipartMessage.
DEFINE OUTPUT PARAMETER messageH AS HANDLE.
```

For information on this procedure in context, see Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

createStreamMessage

In Session Objects

This procedure creates a new StreamMessage:

SYNTAX

```
PROCEDURE createStreamMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.
```

For information on this procedure in context, see the "Factory Procedures" and the "StreamMessage" sections in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

createTextMessage

In Session Objects

This procedure creates a new TextMessage object:

SYNTAX

```
PROCEDURE createTextMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.
```

For information on this procedure in context, see the "Factory Procedures" and "TextMessage" sections in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

createXMLMessage

In Session Objects

This procedure creates a new XMLMessage:

SYNTAX

```
PROCEDURE createXMLMessage.
DEFINE OUTPUT PARAMETER messageHandle AS HANDLE.
```

For information on this procedure in context, see the "Factory Procedures" and "XMLMessage" sections in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

deleteConsumer

In Message Consumer Objects

This procedure ends the life of a Message Consumer object. It cancels the subscription (in the Pub/Sub domain) or the association with a queue (in the PTP domain) and deletes the Message Consumer Object:

SYNTAX

```
PROCEDURE deleteConsumer.
```

NOTE: This procedure executes remotely (sends a message to the SonicMQ Adapter).

For information on this procedure in context, see the "Terminating the Message Consumer" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

deleteMessage

In Message Objects

This procedure deletes a message:

SYNTAX

```
PROCEDURE deleteMessage.
```

For information on this procedure in context, see the "Deleting Messages" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

deleteSession

In Session Objects

This procedure closes a session and its underlying connection and deletes the session procedure:

SYNTAX

```
PROCEDURE deleteSession.
```

NOTE: This procedure executes remotely (sends a message to the SonicMQ Adapter).

For information on this procedure in context, see the "Session-level Methods" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

endOfStream

In Message Objects

This function returns true if the application retrieved the last text segment, the last item of a stream, or the last bytes segment:

SYNTAX

```
FUNCTION endOfStream RETURNS LOGICAL.
```

NOTE: An application should not call endOfStream if it used getMemptr for extracting the data.

For information on this function in context, see the "TextMessage," "StreamMessage," and "BytesMessage" sections in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getAdapterService

In Session Objects

This function returns the value set by the preceding setAdapterService. Null is returned if setAdapterService was not called:

SYNTAX

```
FUNCTION getAdapterService RETURNS CHAR.
```

For information on this function in context, see the "Getting JMS Connection and Session Attributes" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getApplicationContext

In Message Consumer Objects

This function returns application context information:

SYNTAX

```
FUNCTION getApplicationContext RETURNS HANDLE.
```

For information on this function in context, see the "Getting Message Handler Properties" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getBrokerURL

In Session Objects

This function returns the value set by the preceding setBrokerURL. Null is returned if setBrokerURL was not called:

SYNTAX

```
FUNCTION getBrokerURL RETURNS CHAR.
```

For information on this function in context, see the "Getting JMS Connection and Session Attributes" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getBytesCount

In Message Objects

This function returns the number of bytes in a BytesMessage:

SYNTAX

```
FUNCTION getBytesCount RETURNS INT.
```

For information on this function in context, see the "BytesMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getBytesPartByID

In Message Objects

This function expects a MEMPTR and a content ID. If successful, the function retrieves a bytes part and returns the content type as a CHARACTER string.

SYNTAX

```
FUNCTION getBytesPartByID RETURNS CHARACTER
  (INPUT contentID AS INTEGER, OUTPUT memPtr AS MEMPTR).
```

NOTE: Before calling this function, call SET-SIZE to free any memory allocated by the MEMPTR.

NOTE: The bytes part does not undergo any code-page conversion. If it consists of text data, it is encoded in UTF-8. To encode it differently, either convert the code page manually or use one of the "getTextPartBy..." routines.

For information on this function in context, see the "MultipartMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getBytesPartByIndex

In Message Objects

This function expects an index. If successful, the function retrieves a bytes part and returns the content type as a CHARACTER string.

SYNTAX

```
FUNCTION getMessagePartByIndex RETURNS CHARACTER
  (INPUT iIndex AS INTEGER, OUTPUT memPtr AS MEMPTR).
```

NOTE: Before calling this function, call SET-SIZE to free any memory allocated by the MEMPTR.

NOTE: The bytes part does not undergo any code-page conversion. If it consists of text data, it is encoded in UTF-8. To encode it differently, either convert the code page manually or use one of the “getTextPartBy...” routines.

For information on this function in context, see the “MultipartMessage” section in Chapter 13, “Accessing SonicMQ Messaging From the Progress 4GL.”

getBytesToRaw

In Message Objects

This function gets a bytes item from a MapMessage:

SYNTAX

```
FUNCTION getBytesToRaw RETURNS RAW (itemName AS CHAR).
```

For information on this function in context, see the “Getting Item Values By Name In a MapMessage” section in Chapter 13, “Accessing SonicMQ Messaging From the Progress 4GL.”

getChar

In Message Objects

This function gets an item of any data type except bytes from a MapMessage:

SYNTAX

```
FUNCTION getChar RETURNS CHAR (itemName AS CHAR).
```

For information on this function in context, see the "Getting Item Values By Name In a MapMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getCharCount

In Message Objects

This function returns the total number of characters in a message:

SYNTAX

```
FUNCTION getCharCount RETURNS INT.
```

For information on this function in context, see the "TextMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getCharProperty

In Message Objects

This function returns message properties of any data type:

SYNTAX

```
FUNCTION getCharProperty RETURNS CHAR (propertyName AS CHAR).
```

For information on this function in context, see the "Getting Message Properties" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getClientID

In Session Objects

This function returns the value set by the preceding setClientID. Null is returned if setClientID was not called:

SYNTAX

```
FUNCTION getClientID RETURNS CHAR.
```

For information on this function in context, see the "Getting JMS Connection and Session Attributes" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getConnectionID

In Session Objects

This function returns the AppServer connection ID. This value is typically used to correlate the session to log entries on the server side. It returns UNKNOWN when called before calling beginSession:

SYNTAX

```
FUNCTION getConnectionID RETURNS CHAR.
```

For information on this function in context, see the "Session-level Methods" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getConnectionMetaData

In Session Objects

This function returns a comma-separated list of connection and provider attributes in this order: JMSVersion, JMSMajorVersion, JMSMinorVersion, JMSProviderName, JMSProviderVersion, JMSProviderMajorVersion, and JMSProviderMinorVersion:

SYNTAX

```
FUNCTION getConnectionMetaData RETURNS CHAR.
```

For information on this procedure in context, see the "Getting JMS Connection and Session Attributes" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getConnectionURLs

In Session Objects

This function returns a comma-separated list of Sonic Broker URLs that the client will try to connect to:

SYNTAX

```
FUNCTION getConnectionURLs RETURNS CHARACTER.
```

For information on this routine in context, see the "Fail-over Support" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getDecimal

In Message Objects

This function gets any numeric item from a MapMessage:

SYNTAX

```
FUNCTION getDecimal RETURNS DECIMAL (itemName AS CHAR).
```

For information on this function in context, see the "Getting Item Values By Name In a MapMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getDecimalProperty

In Message Objects

This function returns any numeric message property:

SYNTAX

```
FUNCTION getDecimalProperty RETURNS DECIMAL (propertyName AS CHAR).
```

For information on this function in context, see the "Getting Message Properties" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getDefaultPersistency

In Session Objects

This function returns the value specified by setDefaultPersistency. It returns PERSISTENT if setDefaultPersistency was not called:

SYNTAX

```
FUNCTION getDefaultPersistency RETURNS CHAR.
```

For information on this function in context, see the "Getting Message Delivery Parameters" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getDefaultPriority

In Session Objects

This function returns the value specified by setDefaultPriority. It returns 4 if setDefaultPriority was not called:

SYNTAX

```
FUNCTION getDefaultPriority RETURNS INT.
```

For information on this function in context, see the "Getting Message Delivery Parameters" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getDefaultTimeToLive

In Session Objects

This function returns the value specified by setDefaultTimeToLive. It returns UNKNOWN if setDefaultTimeToLive was not called:

SYNTAX

```
FUNCTION getDefaultTimeToLive RETURNS DECIMAL.
```

For information on this function in context, see the "Getting Message Delivery Parameters" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getDestinationName

In Message Consumer Objects

This function returns the name of the destination that messages arrive from when the message consumer was passed to subscribe or receiveFromQueue:

SYNTAX

```
FUNCTION getDestinationName RETURNS CHAR.
```

For information on this function in context, see the "Getting Message Handler Properties" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getInt

In Message Objects

This function gets int, short, or bytes items from a MapMessage:

SYNTAX

```
FUNCTION getInt RETURNS INT (itemName AS CHAR).
```

For information on this function in context, see the "Getting Item Values By Name In a MapMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getIntProperty

In Message Objects

This function returns int, short, and byte message properties:

SYNTAX

```
FUNCTION getIntProperty RETURNS INT (propertyName AS CHAR).
```

For information on this function in context, see the "Getting Message Handler Properties" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getItemType

In Message Objects

This function returns the data type of an item in a MapMessage. It returns UNKNOWN if the item does not exist:

SYNTAX

```
FUNCTION getItemType RETURNS CHAR (itemName AS CHAR).
```

For information on this function in context, see the "MapMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getJMSCorrelationID

In Message Objects

This function returns the correlationID. This value is application-defined, typically the ID of the message replied to:

SYNTAX

```
FUNCTION getJMSCorrelationID RETURNS CHAR.
```

For information on this function in context, see the "Getting Message Header Information" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getJMSCorrelationIDAsBytes

In Message Objects

This function returns a proprietary (JMS provider-dependent) correlation ID. When accessing SonicMQ, the bytesCorrelationID field can be used for storing application-defined values:

SYNTAX

```
FUNCTION getJMSCorrelationIDAsBytes RETURNS RAW.
```

For information on this function in context, see the "Getting Message Header Information" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getJMSDeliveryMode

In Message Objects

This function returns the delivery mode. This value is PERSISTENT, NON_PERSISTENT, or DISCARDABLE. The message receiver never gets the NON_PERSISTENT_ASYNC value. A message sent using NON_PERSISTENT_ASYNC is received with the standard NON_PERSISTENT value:

SYNTAX

```
FUNCTION getJMSDeliveryMode RETURNS CHAR.
```

For information on this function in context, see the "Getting Message Header Information" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getJMSDestination

In Message Objects

This function returns the name of the destination this message was sent to. The value is valid after the message was sent (at the sender side) and in the received message (at the receiver side):

SYNTAX

```
FUNCTION getJMSDestination RETURNS CHAR.
```

For information on this function in context, see the "Getting Message Header Information" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getJMSExpiration

In Message Objects

This function returns the expiration time (GMT):

SYNTAX

```
FUNCTION getJMSExpiration RETURNS DECIMAL.
```

For information on this function in context, see the "Getting Message Header Information" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getJMSMessageID

In Message Objects

This function returns the message ID, a unique ID that the JMS server assigns to each message:

SYNTAX

```
FUNCTION getJMSMessageID RETURNS CHAR.
```

For information on this function in context, see the "Getting Message Header Information" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getJMSPriority

In Message Objects

This function returns priority values in the range of 0–9:

SYNTAX

```
FUNCTION getJMSPriority RETURNS int.
```

For information on this function in context, see the "Getting Message Header Information" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getJMSRedelivered

In Message Objects

This function returns true (at the receiver side) if this is not the first delivery of this message. A second delivery can take place if the first delivery is not acknowledged by the receiver or the transaction was rolled back, in a transacted session:

SYNTAX

```
FUNCTION getJMSRedelivered RETURNS LOGICAL.
```

For information on this function in context, see the "Getting Message Header Information" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getJMSReplyTo

In Message Objects

This function returns the reply destination. The destination can be the name of a queue, even if the message is received from a Pub/Sub session, and the destination can be the name of a topic even if the message is received from a PTP session. An application must call getReplyToDestinationType if both a queue destination and a topic destination might be stored in the received message:

SYNTAX

```
FUNCTION getJMSReplyTo RETURNS CHAR.
```

For information on this function in context, see the "Getting Message Header Information" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getJMSServerName

In Session Objects

This function returns the value set by the preceding setJMSServerName. Null is returned if setJMSServerName was not called:

SYNTAX

```
FUNCTION getJMSServerName RETURNS CHAR.
```

For information on this function in context, see the "Getting JMS Connection and Session Attributes" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getJMSTimestamp

In Message Objects

This function returns the message sending time, which is the difference in milliseconds, between the message creation time and midnight, January 1, 1970 UTC:

SYNTAX

```
FUNCTION getJMSTimestamp RETURNS DECIMAL.
```

For information on this function in context, see the "Getting Message Header Information" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getJMSType

In Message Objects

This function returns a proprietary (JMS provider-dependent) type name. When accessing SonicMQ, the JMSType field can be used for storing application-defined values:

SYNTAX

```
FUNCTION getJMSType RETURNS CHAR.
```

For information on this function in context, see the "Getting Message Header Information" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getLoadBalancing

In Session Objects

This function returns a LOGICAL value indicating whether load balancing is enabled — that is, whether the client is willing to have a connect request redirected to another broker within a SonicMQ cluster. TRUE indicates load balancing is enabled. FALSE indicates it it not enabled:

SYNTAX

```
FUNCTION getLoadBalancing RETURNS LOGICAL.
```

For information on this function in context, see the "Load Balancing" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getLogical

In Message Objects

This function returns a boolean item by name from a MapMessage:

SYNTAX

```
FUNCTION getLogical RETURNS LOGICAL (itemName AS CHAR).
```

For information on this function in context, see the "Getting Item Values By Name In a MapMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getLogicalProperty

In Message Objects

This function returns a boolean message property:

SYNTAX

```
FUNCTION getLogicalProperty RETURNS LOGICAL (propertyName AS CHAR).
```

For information on this function in context, see the "Getting Message Properties" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getMapNames

In Message Objects

This function returns a comma-separated list of the item names in a MapMessage:

SYNTAX

```
FUNCTION getMapNames RETURNS CHAR.
```

For information on this function in context, see the "MapMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getMEMPTR

In Message Objects

This function returns a reference to a MEMPTR variable that contains exactly all the bytes of a BytesMessage. This function implicitly calls reset. If the message was in a write-only mode, it will be in a read-only/reset mode after the call. The getMemptr function does not create a copy of the MEMPTR variable; it returns a reference to the data maintained by the Message object. The deleteMessage call releases the variable's memory, and the caller must copy any data it needs or needs to modify before deleting the message:

SYNTAX

```
FUNCTION getMemptr RETURNS MEMPTR.
```

For information on this function in context, see the "Writing and Reading MEMPTR Bytes Data" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getMessagePartByID

In Message Objects

This function expects a content ID. If successful, the function retrieves a message part and returns the content type as a character string.

SYNTAX

```
FUNCTION getMessagePartByID RETURNS CHARACTER
  (INPUT contentID AS INTEGER, OUTPUT messageParth AS HANDLE).
```

NOTE: When you use the same handle variable to retrieve multiple message parts, after each retrieval, call deleteMessage on the handle variable to free the message part.

For information on this function in context, see the "MultipartMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getMessagePartByIndex

In Message Objects

This function expects an index. If successful, the function retrieves a handle to the message part and returns the content type as a character string.

SYNTAX

```
FUNCTION getMessagePartByIndex RETURNS CHARACTER
  (INPUT index AS INTEGER, OUTPUT messageParth AS HANDLE).
```

NOTE: When you use the same handle variable to retrieve multiple message parts, between retrievals, call deleteMessage on the handle variable to free the message part.

For information on this function in context, see the "MultipartMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getMessageType

In Message Objects

This function returns the SonicMQ Adapter message type: TextMessage, MapMessage, StreamMessage, BytesMessage, HeaderMessage, XMLMessage, or MultipartMessage:

SYNTAX

```
FUNCTION getMessageType RETURNS CHAR.
```

For information on this function in context, see the "Getting Message Header Information" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getNoAcknowledge

In Message Consumer Objects

This function returns true if setNoAcknowledge was called:

SYNTAX

```
FUNCTION getNoAcknowledge RETURNS LOGICAL.
```

For information on this function in context, see the "Stopping Acknowledgement" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getPartCount

In Message Objects

This function returns the number of parts in a multipart message:

SYNTAX

```
FUNCTION getPartCount RETURNS INTEGER.
```

For information on this function in context, see the "MultipartMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getPassword

In Session Objects

This function returns the value set by the preceding setPassword. Null is returned if setPassword was not called:

SYNTAX

```
FUNCTION getPassword RETURNS CHAR.
```

For information on this procedure in context, see the "Getting JMS Connection and Session Attributes" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getProcHandle

In Message Consumer Objects

This function returns the handle to a procedure that contains the name of an internal procedure for handling messages:

SYNTAX

```
FUNCTION getProcHandle RETURNS HANDLE.
```

For information on this function in context, see the "Getting Message Handler Properties" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getProcName

In Message Consumer Objects

This function returns the name of the internal procedure for handling messages:

SYNTAX

```
FUNCTION getProcName RETURNS CHAR.
```

For information on this function in context, see the "Getting Message Handler Properties" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getPropertyNames

In Message Objects

This function returns a comma-separated list of the properties of a message:

SYNTAX

```
FUNCTION getPropertyNames RETURNS CHAR.
```

For information on this function in context, see the "Getting Message Properties" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getPropertyType

In Message Objects

This function returns the message property's data type. UNKNOWN is returned if the property was not set in the message:

SYNTAX

```
FUNCTION getPropertyType RETURNS CHAR (propertyName AS CHAR).
```

For information on this function in context, see the "Getting Message Properties" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getReplyAutoDelete

In Message Consumer Objects

This function returns the value set by setReplyAutoDelete:

SYNTAX

```
FUNCTION getReplyAutoDelete RETURNS LOGICAL.
```

For information on this function in context, see the "Getting Message Handler Properties" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getReplyPersistency

In Message Consumer Objects

This function returns the setReplyPersistency value; it is PERSISTENT if setReplyPersistency was not called:

SYNTAX

```
FUNCTION getReplyPersistency RETURNS CHAR.
```

For information on this function in context, see the "Getting Message Handler Properties" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getReplyPriority

In Message Consumer Objects

This function returns the setReplyPriority value; it is 4 if setReplyPriority was not called:

SYNTAX

```
FUNCTION getReplyPriority RETURNS INT.
```

For information on this function in context, see the "Getting Message Handler Properties" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getReplyTimeToLive

In Message Consumer Objects

This function returns the setReplyTimeToLive value; it is UNKNOWN if setReplyTimeToLive was not called:

SYNTAX

```
FUNCTION getReplyTimeToLive RETURNS DECIMAL.
```

For information on this function in context, see the "Getting Message Handler Properties" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getReplyToDestinationType

In Message Objects

This function returns "queue," "topic," or UNKNOWN. The UNKNOWN value is returned if the message was just created, not sent yet, and setReplyToDestinationType was not called. Applications use this function when the domain of the ReplyTo field is not known:

SYNTAX

```
FUNCTION getReplyToDestinationType RETURNS CHAR.
```

For information on this function in context, see the "Getting Message Header Information" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getReuseMessage

In Message Consumer Objects

This function returns true if setReuseMessage was called; if not, it returns false:

SYNTAX

```
FUNCTION getReuseMessage RETURNS LOGICAL.
```

For information on this function in context, see the "Specifying Message Reuse" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getSession

In Message Consumer Objects

This function returns a handle to the session:

SYNTAX

```
FUNCTION getSession RETURNS HANDLE.
```

For information on this function in context, see the "Getting Message Handler Properties" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getSequential

In Session Objects

This function returns a LOGICAL value indicating how a fail-over list is used — that is, whether clients try to connect to brokers in a connection list sequentially or randomly.

TRUE indicates sequentially. FALSE indicates randomly:

SYNTAX

```
FUNCTION getSequential RETURNS LOGICAL.
```

For information on this function in context, see the "Fail-over Support" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getSingleMessageAcknowledgement

In Session Objects

This function indicates whether a client session is configured to use single-message acknowledgement.

If the function returns TRUE, the client session is configured to use single-message acknowledgement. If the function returns FALSE, the client session is not so configured:

SYNTAX

```
FUNCTION getSingleMessageAcknowledegment RETURNS LOGICAL.
```

For information on this function in context, see the "Single Message Acknowledgement" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getText

In Message Objects

This function returns all the text in the message and then implicitly calls reset:

SYNTAX

```
FUNCTION getText RETURNS CHAR.
```

NOTE: A run-time error occurs if the size of the message is too large to be handled by the 4GL interpreter.

For information on this function in context, see the "Getting Text" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getTextPartByID

In Message Objects

This function expects a content ID. If successful, the function retrieves a text part and returns the content type as a CHARACTER string.

This function converts the text part from UTF-8 to the SESSION:CPINTERNAL code page.

SYNTAX

```
FUNCTION getTextPartByID RETURNS CHARACTER
  (INPUT contentID AS INTEGER, OUTPUT partBody AS CHARACTER).
```

NOTE: If the message body exceeds 32K, this function raises an error. To avoid this, use getBytesPartByID.

For information on this function, see the "MultipartMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getTextPartByIndex

In Message Objects

This function expects an index. If successful, the function retrieves a text part and returns the content type as a CHARACTER string.

This function converts the text part from UTF-8 to the SESSION:CPINTERNAL code page.

SYNTAX

```
FUNCTION getTextPartByIndex RETURNS CHARACTER
  (INPUT iIndex AS INTEGER, OUTPUT partBody AS CHARACTER).
```

NOTE: If the message body exceeds 32K, this function raises an error. To avoid this, use getBytesPartByIndex.

For information on this function in context, see the "MultipartMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getTextSegment

In Message Objects

This function can be called in read-only mode to return the next text segment when handling large messages:

SYNTAX

```
FUNCTION getTextSegment RETURNS CHAR.
```

For information on this function in context, see the "Getting Text" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getTransactedReceive

In Session Objects

This function returns the value set by the preceding setTransactedReceive. False is returned if setTransactedReceive was not called:

SYNTAX

```
FUNCTION getTransactedReceive RETURNS LOGICAL.
```

For information on this function in context, see the "Getting JMS Connection and Session Attributes" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getTransactedSend

In Session Objects

This function returns the value set by the preceding setTransactedSend. False is returned if setTransactedSend was not called:

SYNTAX

```
FUNCTION getTransactedSend RETURNS LOGICAL.
```

For information on this function in context, see the "Getting JMS Connection and Session Attributes" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

getUser

In Session Objects

This function returns the value set by the preceding setUser. Null is returned if setUser was not called:

SYNTAX

```
FUNCTION getUser RETURNS CHAR.
```

For information on this procedure in context, see the "Getting JMS Connection and Session Attributes" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

hasReplyTo

In Message Objects

This function returns true if the JMSreplyTo header was set:

SYNTAX

```
FUNCTION hasReplyTo RETURNS LOGICAL.
```

For information on this function in context, see the "Getting Message Header Information" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

inErrorHandling

In Message Consumer Objects

This function returns true when called from a message handler if the application is handling an error message:

SYNTAX

```
FUNCTION inErrorHandling RETURNS LOGICAL.
```

For information on this function in context, see the "Getting Message Handler Properties" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

inMessageHandling

In Message Consumer Objects

This function returns true when called from a message handler if the application is handling the data in a subscription (or queue) message:

SYNTAX

```
FUNCTION inMessageHandling RETURNS LOGICAL.
```

For information on this function in context, see the "Getting Message Handler Properties" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

inQueueBrowsing

In Message Consumer Objects

This function returns true when called from a message handler if an application is handling a queue browsing message:

SYNTAX

```
FUNCTION inQueueBrowsing RETURNS LOGICAL.
```

For information on this function in context, see the "Getting Message Handler Properties" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

inReplyHandling

In Message Consumer Objects

This function returns true when called from a message handler if an application is handling a reply message:

SYNTAX

```
FUNCTION inReplyHandling RETURNS LOGICAL.
```

For information on this function in context, see the "Getting Message Handler Properties" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

isMessagePart

In Message Objects

This function expects two input parameters: a handle to a message part, and an index. The function returns TRUE if the part specified by the index is a SonicMQ message. This lets you know whether to use getMessagePart or one of the other message-part access methods.

SYNTAX

```
FUNCTION isMessagePart RETURNS LOGICAL
  (INPUT messageH AS HANDLE, INPUT index AS INTEGER).
```

For information on this function in context, see the "MultipartMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

JMS–MAXIMUM–MESSAGES

This global variable changes the maximum number of JMS messages in a 4GL session; the default is 50 (the total number of messages created by the application plus messages received from JMS). To change the default to *new–val*, the following definition must be included in the main procedure of the 4GL application:

SYNTAX

```
DEFINE NEW GLOBAL SHARED VAR JMS-MAXIMUM-MESSAGES AS INT INIT new-val.
```

For information on this global variable in context, see the "Maximum Number Of Messages" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

messageHandler

In Message Objects

This procedure handles incoming JMS and error messages. The message handler is written by an application and must be registered with a Message Consumer object. When a message is received, the message handler is called automatically so the application can process the message:

SYNTAX

```
PROCEDURE messageHandler.
DEFINE INPUT PARAMETER message AS HANDLE.
DEFINE INPUT PARAMETER messageConsumer AS HANDLE.
DEFINE OUTPUT PARAMETER reply AS HANDLE.
```

For information on this procedure and its parameters in context, see the "Message Handler" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

moveToNext

In Message Objects

This function moves the cursor to the next data item in a StreamMessage and returns its data type, one of the following values: UNKNOWN, boolean, byte, short, char, int, long, float, double, string, or bytes. UNKNOWN is returned when the value of the item is NULL. When the message is received or after reset is called, the cursor is set before the first data item. It is an error to try to move the cursor beyond the last item:

SYNTAX

```
FUNCTION moveToNext RETURNS CHAR.
```

For information on this function in context, see the "Reading Data From a StreamMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

publish

In Session Objects

This procedure publishes a message to topicName. If the publication is in reply to a received message, topicName can be the replyTo field obtained from the original message:

SYNTAX

```
PROCEDURE publish.
DEFINE INPUT PARAMETER topicName AS CHAR.
DEFINE INPUT PARAMETER message AS HANDLE.
DEFINE INPUT PARAMETER priority AS INT.            /*Session default is
                                                     used if UNKNOWN.*/
DEFINE INPUT PARAMETER timeToLive AS DECIMAL.      /*Session default is
                                                     used if UNKNOWN.*/
DEFINE INPUT PARAMETER deliveryMode AS CHAR.       /*Session default is
                                                     used if UNKNOWN.*/
```

NOTE: This procedure executes remotely (sends a message to the SonicMQ Adapter).

For information on this procedure in context, see the "Publishing To a Topic" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

readBytesToRaw

In Message Objects

This function returns bytes data from the body of a StreamMessage.

It can be called in read-only mode to return the next bytes segment in a BytesMessage. The size of all the byte segments other than the last one is 8192; the size of the last one is 8192 or less:

SYNTAX

```
FUNCTION readBytesToRaw RETURNS RAW.
```

For information on this function in context, see the "Reading Data From a StreamMessage" and "Overcoming the 32K RAW Bytes Limit" sections in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

readChar

In Message Objects

This function returns any message data except bytes data from the body of a StreamMessage:

SYNTAX

```
FUNCTION readChar RETURNS CHAR.
```

For information on this function in context, see the "Reading Data From a StreamMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

readDecimal

In Message Objects

This function returns any numeric data from the body of a StreamMessage:

SYNTAX

```
FUNCTION readDecimal RETURNS DECIMAL.
```

For information on this function in context, see the "Reading Data From a StreamMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

readInt

In Message Objects

This function returns int, short, or bytes data from the body of a StreamMessage:

SYNTAX

```
FUNCTION readInt RETURNS INT.
```

For information on this function in context, see the "Reading Data From a StreamMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

readLogical

In Message Objects

This function returns boolean data from the body of a StreamMessage:

SYNTAX

```
FUNCTION readLogical RETURNS LOGICAL.
```

For information on this function in context, see the "Reading Data From a StreamMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

receiveFromQueue

In Session Objects

This procedure receives messages from `queueName`. The messages are handled asynchronously by the `messageConsumer` procedure:

SYNTAX

```
PROCEDURE receiveFromQueue.
DEFINE INPUT PARAMETER queueName AS CHAR.
DEFINE INPUT PARAMETER messageSelector AS CHAR.   /*Optional. If UNKNOWN,
                                                    receives all messages.*/
DEFINE INPUT PARAMETER messageConsumer AS HANDLE.
```

NOTE: This procedure executes remotely (sends a message to the SonicMQ Adapter).

For information on this procedure in context, see the "Receiving Messages From a Queue" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

recover

In Session Objects

This procedure starts redelivering all unacknowledged messages received up to that point in the current session. It is an error to call this method if the session is transacted for receiving:

SYNTAX

```
PROCEDURE recover.
```

NOTE: This procedure executes remotely (sends a message to the SonicMQ Adapter).

For information on this procedure in context, see the "Transaction and Recovery Methods" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

requestReply

In Session Objects

Request/Reply is a mechanism for the JMSReplyTo message header field to specify the destination where a reply to a message should be sent. The requestReply method sends a message to a destination and designates the messageConsumer parameter for processing replies. Since this call is supported by both the `pubsubsession.p` object and the `ptpsession.p` object, Progress uses the term, destination, which can be used for both topics and queues.

The 4GL–JMS implementation automates the request/reply sequence:

- Sending a reply by setting the reply OUTPUT parameter of the message handler
- Requesting a reply by calling the requestReply session method with a reply message consumer

The 4GL–JMS implementation uses a temporary destination for the reply. It is an error to set the JMSReplyTo field of the message explicitly if requestReply is used. The reply is received by messageConsumer asynchronously, just like any other message reception. The temporary destination is deleted when the Message Consumer object is deleted:

SYNTAX

```
PROCEDURE requestReply.
DEFINE INPUT PARAMETER destination AS CHAR.
DEFINE INPUT PARAMETER message AS HANDLE.
DEFINE INPUT PARAMETER replySelector AS CHAR.      /*UNKNOWN means
                                                     receiving all replies*/
DEFINE INPUT PARAMETER messageConsumer AS HANDLE.  /*UNKNOWN is illegal*/
DEFINE INPUT PARAMETER priority AS INT.            /*Session default is
                                                     used if UNKNOWN./*
DEFINE INPUT PARAMETER timeToLive AS DECIMAL.      /*Session default is
                                                     used if UNKNOWN./*
DEFINE INPUT PARAMETER deliveryMode AS CHAR.       /*Session default is
                                                     used if UNKNOWN./*
```

NOTE: This procedure executes remotely (sends a message to the SonicMQ Adapter).

For information on this procedure in context, see the "Request/Reply" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

reset

In Message Objects

This procedure changes the mode of a message from write-only to read-only mode and positions the cursor before the first segment. Sending the message causes an implicit reset and the message becomes read-only. The message arrives at the receiver in a reset state:

SYNTAX

```
PROCEDURE reset.
```

For information on this procedure in context, see the "TextMessage," "StreamMessage," and "BytesMessage" sections in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

rollbackReceive

In Session Objects

This procedure starts redelivering the messages received up to that point in the current transaction. It is an error to call this procedure in a Session object that is not transacted for receiving:

SYNTAX

```
PROCEDURE rollbackReceive.
```

NOTE: This procedure executes remotely (sends a message to the SonicMQ Adapter).

For information on this procedure in context, see the "Transaction and Recovery Methods" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

rollbackSend

In Session Objects

This procedure discards all messages sent up to that point in the current transaction. It is an error to call this method in a Session object that is not transacted for sending:

SYNTAX

```
PROCEDURE rollbackSend.
```

NOTE: This procedure executes remotely (sends a message to the SonicMQ Adapter).

For information on this procedure in context, see the "Transaction and Recovery Methods" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

sendToQueue

In Session Objects

This procedure sends a message to queueName. If the sending is in reply to a received message, queueName can be the replyTo field obtained from the original message:

SYNTAX

```
PROCEDURE sendToQueue.
DEFINE INPUT PARAMETER queueName AS CHAR.
DEFINE INPUT PARAMETER message AS HANDLE.
DEFINE INPUT PARAMETER priority AS INT.              /*Session default is
                                                       used if UNKNOWN.*/
DEFINE INPUT PARAMETER timeToLive AS DECIMAL.        /*Session default is
                                                       used if UNKNOWN.*/
DEFINE INPUT PARAMETER deliveryMode AS CHAR.         /*Session default is
                                                       used if UNKNOWN.*/
```

NOTE: This procedure executes remotely (sends a message to the SonicMQ Adapter).

For information on this procedure in context, see the "Sending Messages To a Queue" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setAdapterService

In Session Objects

This procedure specifies the service name under which the SonicMQ Adapter is registered with the NameServer. The default is `adapter.progress.jms`; if the SonicMQ Adapter uses that service name, setAdapterService is unnecessary:

SYNTAX

```
PROCEDURE setAdapterService.
DEFINE INPUT PARAMTER serviceName AS CHAR,
```

For information on this procedure in context, see the "Setting JMS Connection and Session Attributes" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setApplicationContext

In Message Consumer Objects

This procedure passes context to the message handler. The handler parameter is typically a handle to a persistent procedure implemented by the application. When the message handler is called, it gets that handler and uses it, for example, to deposit error information in the application's context by calling a specific handler's internal procedure:

SYNTAX

```
PROCEDURE setApplicationContext.
DEFINE INPUT PARAMETER handler AS HANDLE.
```

For information on this procedure in context, see the "Setting Context" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setBoolean

In Message Objects

This procedure converts data to the JMS boolean data type in a MapMessage:

SYNTAX

```
PROCEDURE setBoolean.
DEFINE INPUT PARAMETER itemName AS CHAR.
DEFINE INPUT PARAMETER value AS LOGICAL.
```

For information on this procedure in context, see the "Setting Items In a MapMessage" section in Appendix 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setBooleanProperty

In Message Objects

This procedure sets a boolean message property. An UNKNOWN value is considered a false value:

SYNTAX

```
PROCEDURE setBooleanProperty.
DEFINE INPUT PARAMETER propertyName AS CHAR.
DEFINE INPUT PARAMETER propertyValue AS LOGICAL.
```

For information on this procedure in context, see the "Setting Message Properties" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setBrokerURL

In Session Objects

This procedure sets the value of the SonicMQ broker URL. If set on the client, it overwrites the default brokerURL property set on the SonicMQ Adapter side. The creation of a session fails if no value is set on the client or at the SonicMQ Adapter:

SYNTAX

```
PROCEDURE setBrokerURL.
DEFINE INPUT PARAMETER brokerURL AS CHAR.
```

For information on this procedure in context, see the "Setting JMS Connection and Session Attributes" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setByte

In Message Objects

This procedure converts data in a MapMessage to the JMS byte data type; byte values are –128 to 127. The server returns a NumberFormatException message for a value overflow:

SYNTAX

```
PROCEDURE setByte.
DEFINE INPUT PARAMETER itemName AS CHAR.
DEFINE INPUT PARAMETER value AS INT.
```

For information on this procedure in context, see the "Setting Items In a MapMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setByteProperty

In Message Objects

This procedure sets a bytes property in a message; the values range from –128 to 127. The server returns a NumberFormatException message for a value overflow:

SYNTAX

```
PROCEDURE setByteProperty.
DEFINE INPUT PARAMETER propertyName AS CHAR.
DEFINE INPUT PARAMETER propertyValue AS INT.
```

For information on this procedure in context, see the "Setting Message Properties" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setBytesFromRaw

In Message Objects

This procedure converts data in a MapMessage to the JMS bytes data type:

SYNTAX

```
PROCEDURE setBytesFromRaw.
DEFINE INPUT PARAMETER itemName AS CHAR.
DEFINE INPUT PARAMTER values AS RAW.
```

For information on this procedure in context, see the "Setting Items In a MapMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setChar

In Message Objects

This procedure converts data in a MapMessage to the JMS char data type; the number of characters in the char value must be one:

SYNTAX

```
PROCEDURE setChar.
DEFINE INPUT PARAMETER itemName AS CHAR.
DEFINE INPUT PARAMETER value AS CHAR.
```

For information on this procedure in context, see the "Setting Items In a MapMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setClientID

In Session Objects

This procedure sets the clientID value for the SonicMQ broker connection and overwrites the default clientID set on the server side. A clientID is required for durable subscriptions:

SYNTAX

```
PROCEDURE setClientID.
DEFINE INPUT PARAMETER clientID AS CHAR.
```

For information on this procedure in context, see the "Setting JMS Connection and Session Attributes" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setConnectionURLs

In Session Objects

This procedure specifies a list of broker URLs for the client to try to connect to. *brokerList* is a comma-separated list of Sonic Broker URLs. If *brokerList* is not set to ? (the unknown value), it overrides the URL specified by setBrokerURL.

Call this procedure instead of setBrokerURL when there is a list of broker URLs:

SYNTAX

```
PROCEDURE setConnectionURLs.
DEFINE INPUT PARAMETER brokerList AS CHARACTER.
```

For information on this procedure in context, see the "Fail-over Support" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setDefaultPersistency

In Session Objects

This procedure sets the default message persistency value for all messages sent in that session. The values are: PERSISTENT, NON_PERSISTENT, NON_PERSISTENT_ASYNC, DISCARDABLE (which applies only when publishing to a topic), and UNKNOWN (?). The default value is PERSISTENT. The evaluation is case insensitive. A call with an UNKNOWN value has no effect. NON_PERSISTENT_ASYNC is a SonicMQ extension of the JMS specification.

If DISCARDABLE is used when publishing other than to a topic, an error is raised:

SYNTAX

```
PROCEDURE setDefaultPersistency.
DEFINE INPUT PARAMETER deliveryMode AS CHAR.
```

For information on this procedure in context, see the "Setting Message Delivery Parameters" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setDefaultPriority

In Session Objects

This procedure sets the default message priority for all messages sent in that session. The range of priority values is 0–9; the default is 4. Setting an UNKNOWN value has no effect:

SYNTAX

```
PROCEDURE setDefaultPriority.
DEFINE INPUT PARAMETER priority AS INT.
```

For information on this procedure in context, see the "Setting Message Delivery Parameters" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setDefaultTimeToLive

In Session Objects

This procedure sets the default time to live, the number of milliseconds from the time a message is sent to the time the JMS server can delete the message from the system. A setting of 0 specifies that the message never expires. The default is JMS broker-dependent; the Progress SonicMQ default value is 0. Any fractional part of the decimal value is truncated. If the value does not fit in a Java long value, Java rules for decimal-to-long conversions are used. Setting an UNKNOWN value has no effect:

SYNTAX

```
PROCEDURE setDefaultTimeToLive.
DEFINE INPUT PARAMETER millis AS DECIMAL.
```

For information on this procedure in context, see the "Setting Message Delivery Parameters" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setDouble

In Message Objects

This procedure converts data in a MapMessage to the JMS double data type:

SYNTAX

```
PROCEDURE setDouble.
DEFINE INPUT PARAMETER itemName AS CHAR.
DEFINE INPUT PARAMETER value AS DECIMAL.
```

For information on this procedure in context, see the "Setting Items In a MapMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setDoubleProperty

In Message Objects

This procedure sets a double message property:

SYNTAX

```
PROCEDURE setDoubleProperty.
DEFINE INPUT PARAMETER propertyName AS CHAR.
DEFINE INPUT PARAMETER propertyValue AS DECIMAL.
```

For information on this procedure in context, see the "Setting Message Properties" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setErrorHandler

In Session Objects

To handle asynchronous conditions programmatically, applications create a messageConsumer object and pass it to the setErrorHandler procedure in the Session object. Asynchronous conditions are always reported as a TextMessage with several possible CHAR message properties. The CHAR properties that might be included in the message header are: exception, errorCode, linkedException-1, linkedException-2… linkedException-N (where N is a number of additional exceptions linked to the main exception).

The getPropertyNames message function can be used to get the list of properties in the error message header. If the application does not call setErrorHandler, a default error handler displays the error message and the properties in alert boxes:

SYNTAX

```
PROCEDURE setErrorHandler.
DEFINE INPUT PARAMETER messageConsumer AS HANDLE.
```

NOTE: The application must create the error-handling messageConsumer object and call setErrorHandler after calling beginSession.

For information on this procedure in context, see the "Error Handling" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."For an example, see the "Installing an Error Handler To Handle an Asynchronous Error" section in Chapter 14, "Using the SonicMQ Adapter."

setFloat

In Message Objects

This procedure converts data in a MapMessage to the JMS float data type:

SYNTAX

```
PROCEDURE setFloat.
DEFINE INPUT PARAMETER itemName AS CHAR.
DEFINE INPUT PARAMETER value AS DECIMAL.
```

For information on this procedure in context, see the "Setting Items In a MapMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setFloatProperty

In Message Objects

This procedure sets a float message property:

SYNTAX

```
PROCEDURE setFloatProperty.
DEFINE INPUT PARAMETER propertyName AS CHAR.
DEFINE INPUT PARAMETER propertyValue AS DECIMAL.
```

For information on this procedure in context, see the "Setting Message Properties" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setInt

In Message Objects

This procedure converts data in a MapMessage to the JMS int data type:

SYNTAX

```
PROCEDURE setInt.
DEFINE INPUT PARAMETER itemName AS CHAR.
DEFINE INPUT PARAMETER value AS INT.
```

For information on this procedure in context, see the "Setting Items In a MapMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setIntProperty

In Message Objects

This procedure sets a JMS int message property:

SYNTAX

```
PROCEDURE setIntProperty.
DEFINE INPUT PARAMETER propertyName AS CHAR.
DEFINE INPUT PARAMETER propertyValue AS INT.
```

For information on this procedure in context, see the "Setting Message Properties" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setJMSCorrelationID

In Message Objects

This procedure sets the correlationID. This value is application-defined; typically it is set to the ID of the message replied to:

SYNTAX

```
PROCEDURE setJMSCorrelationID
DEFINE INPUT PARAMETER correlationID AS CHAR.
```

For information on this procedure in context, see the "Setting Message Header Information" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setJMSCorrelationIDAsBytes

In Message Objects

This procedure sets the bytes correlationID, a proprietary (JMS provider-dependent) value. When accessing SonicMQ, the bytesCorrelationID field can be used for storing application-defined values:

SYNTAX

```
PROCEDURE setJMSCorrelationIDAsBytes
DEFINE INPUT PARAMETER bytesCorrelationID AS RAW.
```

For information on this procedure in context, see the "Setting Message Header Information" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setJMSReplyTo

In Message Objects

This procedure sets a destination for replies:

SYNTAX

```
PROCEDURE setJMSReplyTo
DEFINE INPUT PARAMETER destination AS CHAR.
```

NOTE: The destination can be a name of a queue even if the message is sent by a Pub/Sub session, and the destination can be the name of the topic even if the message is sent by a PTP session. However, in that case, setReplyToDestinationType must be called to set the correct destination type.

For information on this procedure in context, see the "Setting Message Header Information" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setJMSServerName

In Session Objects

This procedure specifies the JMS broker implementation, SonicMQ. If set on the client, it overwrites the jmsServerName property set on the SonicMQ Adapter side:

SYNTAX

```
PROCEDURE setJmsServerName.
DEFINE INPUT PARAMETER jmsServerName AS CHAR.
```

For information on this procedure in context, see the "Setting JMS Connection and Session Attributes" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setJMSType

In Message Objects

This procedure sets the type name, which is proprietary (JMS provider-dependent). When accessing SonicMQ, the JMSType field can be used for storing application-defined values:

SYNTAX

```
PROCEDURE setJMSType
DEFINE INPUT PARAMETER typeName AS CHAR.
```

For information on this procedure in context, see the "Setting Message Header Information" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setLoadBalancing

In Session Objects

This procedure turns client-side load balancing on or off.

If client-side load balancing is turned on, the client allows redirection to another Sonic broker in the cluster. If it is turned off, the client does not allow redirection.

SYNTAX

```
PROCEDURE setLoadBalancing
DEFINE INPUT PARAMETER loadBalancing AS LOGICAL.
```

NOTE: If beginSession has already been called on the handle when this procedure executes, an error is raised.

For information on this procedure in context, see the "Load Balancing" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setLong

In Message Objects

This procedure sets long data; any fractional part of the DECIMAL value is truncated:

SYNTAX

```
PROCEDURE setLong.
DEFINE INPUT PARAMETER itemName AS CHAR.
DEFINE INPUT PARAMETER value AS DECIMAL.
```

For information on this procedure in context, see the "Setting Items In a MapMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setLongProperty

In Message Objects

This procedure sets a long message property; any fractional part of the DECIMAL value is truncated:

SYNTAX

```
PROCEDURE setLongProperty.
DEFINE INPUT PARAMETER propertyName AS CHAR.
DEFINE INPUT PARAMETER propertyValue AS DECIMAL.
```

For information on this procedure in context, see the "Setting Message Properties" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setMEMPTR

In Message Objects

This procedure sets the specified number of bytes from the MEMPTR variable starting at startIndex (the first byte is 1) in a BytesMessage. The setMemptr procedure implicitly calls clearBody before setting the data and resets after setting the data. Therefore, it can be used whether the message is in a read-only mode or a write-only mode prior to the call. The call makes a copy of the data. Thus, the memptrVal variable is not modified by the 4GL–JMS implementation and can be modified by the 4GL application after the call without corrupting the message:

SYNTAX

```
PROCEDURE setMemptr.
DEFINE INPUT PARAMETER memptrVar AS MEMPTR.
DEFINE INPUT PARAMETER startIndex AS INT.
DEFINE INPUT PARAMETER numBytes AS INT.
```

For information on this procedure in context, see the "Writing and Reading MEMPTR Bytes Data" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setNoAcknowledge

In Message Consumer Objects

This procedure instructs the 4GL–JMS implementation not to acknowledge this message. This call should be made, for example, if the 4GL application fails to use the data in a message and needs to receive the message again. This call is an error if the session is transacted for receiving. If the messageConsumer object is used to handle error messages or for queue browsing, this call has no effect:

SYNTAX

```
PROCEDURE setNoAcknowledge.
```

For information on this procedure in context, see the "Stopping Acknowledgement" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setNoErrorDisplay

In Session Objects and Message Objects

This procedure turns the automatic display of synchronous errors and conditions on and off. The default value is false. If set to true, synchronous errors and conditions are not automatically displayed by the 4GL–JMS implementation. If set to false, synchronous errors and conditions are automatically displayed in alert boxes. The default value is false.

Messages inherit the display/noDisplay property from the session that created them. However, after the message is created, it is independent from the session, and setNoErrorDisplay must be called in the message itself to change the display/noDisplay property:

SYNTAX

```
PROCEDURE setNoErrorDisplay.
DEFINE INPUT PARAMETER noDisplay AS LOGICAL.
```

For information on this procedure in context, see the "Error Handling" and "Error Display"sections in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setPassword

In Session Objects

This procedure sets the password value for the SonicMQ broker login and overwrites the default password property set on the SonicMQ Adapter side:

SYNTAX

```
PROCEDURE setPassword.
DEFINE INPUT PARAMETER password AS CHAR.
```

For information on this procedure in context, see the "Setting JMS Connection and Session Attributes" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setPingInterval

In Session Objects

This procedure specifies the interval in seconds for the JMS Adapter to actively ping the SonicMQ broker so communication failure can be detected promptly. The setPingInterval functionality is a SonicMQ extension. A pingInterval value can also be specified in the srvrStartupParam property of the adapter as a default for all the clients. No pinging is performed by default. setPingInterval must be called before beginSession is called:

SYNTAX

```
PROCEDURE setPingInterval.
DEFINE INPUT PARAMETER interval AS INT.
```

For information on this procedure in context, see the "Setting JMS Connection and Session Attributes" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL." (Also see the *SonicMQ Programming Guide*.)

setPrefetchCount

In ptpsession

This procedure sets the number of messages a Sonic client can retrieve in a single operation from a queue containing multiple messages. The default is three. For example, a count of three means that Sonic client retrieves up to three message from a queue.

SYNTAX

```
PROCEDURE setPrefetchCount.
DEFINE INPUT PARAMETER count AS INTEGER.
```

NOTE: If procedure is called before beginSession is called, an error is raised.

For information on this procedure in context, see the "Flow Control" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setPrefetchThreshold

In ptpsession

This procedure determines when the Sonic client goes back to the broker for more messages. The default is one. For example, a threshold of one means that Sonic does not go back to the broker for more messages until the last message has been delivered.

SYNTAX

```
PROCEDURE setPrefetchThreshold.
DEFINE INPUT PARAMETER threshold AS INTEGER.
```

NOTE: If this procedure is called before beginSession is called, an error is raised.

For information on this procedure in context, see the "Flow Control" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setReplyAutoDelete

In Message Consumer Objects

This procedure specifies whether all reply messages are to be automatically deleted. This procedure sets this reply property when the message consumer is passed to requestReply. If set to true, all reply messages returned through the message handler's OUPUT parameter are automatically deleted after being sent. The default value is false:

SYNTAX

```
PROCEDURE setReplyAutoDelete.
DEFINE INPUT PARAMETER val AS LOGICAL.
```

For information on this procedure in context, see the "Setting Reply Properties" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setReplyPersistency

In Message Consumer Objects

This procedure sets the value for message persistency when the message consumer is passed to requestReply. The values are: PERSISTENT, NON_PERSISTENT, NON_PERSISTENT_ASYNC, and UNKNOWN. The default value is PERSISTENT. The evaluation is case insensitive. A call with an UNKNOWN value has no effect. The replyPersistency value can be set only once. NON_PERSISTENT_ASYNC is a SonicMQ extension:

SYNTAX

```
PROCEDURE setReplyPersistency.
DEFINE INPUT PARAMETER deliveryMode AS CHAR.
```

For information on this procedure in context, see the "Setting Reply Properties" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setReplyPriority

In Message Consumer Objects

This procedure sets the priority of the reply messages when the message consumer is passed to requestReply. The range of values is 0–9; the default is 4:

SYNTAX

```
PROCEDURE setReplyPriority.
DEFINE INPUT PARAMETER priority AS INT.
```

NOTE: This procedure can be called only once.

For information on this procedure in context, see the "Setting Reply Properties" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setReplyTimeToLive

In Message Consumer Objects

This procedure sets the time to live value of the reply messages when the message consumer is passed to requestReply. (Time to live is the number of milliseconds from the time the message is sent to the time the SonicMQ broker can delete the message from the system. A value of 0 means the message never expires.) The default is JMS system-dependent; the SonicMQ default value is 0. The replyTimeToLive values can be set only once. The fractional part of the decimal value is truncated. If the value does not fit in a Java long value, Java rules for decimal-to-long conversion apply:

SYNTAX

```
PROCEDURE setReplyTimeToLive.
DEFINE INPUT PARAMETER millis AS DECIMAL.
```

For information on this procedure in context, see the "Setting Reply Properties" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setReplyToDestinationType

In Message Objects

This procedure sets the type of the destination specified by setJMSReplyTo; the type can be queue or topic. If setReplyToDestinationType is not called, a default type is automatically set when the message is sent, according to the type of the session:

SYNTAX

```
PROCEDURE setReplyToDestinationType
DEFINE INPUT PARAMETER type AS CHAR.
```

For information on this procedure in context, see the "Setting Message Header Information" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setReuseMessage

In Message Consumer Objects

This procedure instructs the Message Consumer object not to create a new message for each received message. Calling setReuseMessage improves performance: if the procedure is not called, the Message Consumer object creates a new message for each received message. A message that is being reused should not be deleted before the session is deleted:

SYNTAX

```
PROCEDURE setReuseMessage.
```

For information on this procedure in context, see the "Specifying Message Reuse" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setSequential

In Session Objects

Sonic lets clients try to connect to brokers in a connection list in two ways:

- Sequentially — starting with the first broker in the list and working sequentially
- Randomly — repeatedly picking a broker randomly

This procedure lets this be determined by the application.

To attempt load balancing, set *seq* to FALSE, which tells clients to try to connect randomly.

The default is TRUE, which tells clients to try to connect sequentially:

SYNTAX

```
PROCEDURE setSequential.
DEFINE INPUT PARAMETER seq AS LOGICAL.
```

For information on this procedure in context, see the "Fail-over Support" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setShort

In Message Objects

This procedure converts data to the JMS short data type in a MapMessage. The server returns a NumberFormatException message for a value overflow:

SYNTAX

```
PROCEDURE setShort.
DEFINE INPUT PARAMETER itemName AS CHAR.
DEFINE INPUT PARAMETER value AS INT.
```

For information on this procedure in context, see the "Setting Items In a MapMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setShortProperty

In Message Objects

This procedure sets a short message property.The server returns a NumberFormatException message for a value overflow:

SYNTAX

```
PROCEDURE setShortProperty.
DEFINE INPUT PARAMETER propertyName AS CHAR.
DEFINE INPUT PARAMETER propertyValue AS INT.
```

For information on this procedure in context, see the "Setting Message Properties" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setSingleMessageAcknowledgement

In Session Objects

This procedure lets an application turn on single-message acknowledgement for a client session.

If a session is configured to use single-message acknowledgement, the following apply:

- Groups of messages cannot be acknowledged in one operation.
- Acknowledge-and-forward can be used.

SYNTAX

```
PROCEDURE setSingleMessageAcknowledgement.
DEFINE INPUT PARAMETER ackMethod as LOGICAL.
```

NOTE: This procedure must be run before the client session starts. Otherwise, an error is raised.

For information on this procedure in context, see the "Single Message Acknowledgement" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setString

In Message Objects

This procedure converts data in a MapMessage to the JMS String data type:

SYNTAX

```
PROCEDURE setString.
DEFINE INPUT PARAMETER itemName AS CHAR.
DEFINE INPUT PARAMETER value AS CHAR.
```

For information on this procedure in context, see the "Setting Items In a MapMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setStringProperty

In Message Objects

This procedure sets a String message property:

SYNTAX

```
PROCEDURE setStringProperty.
DEFINE INPUT PARAMETER propertyName AS CHAR.
DEFINE INPUT PARAMETER propertyValue AS CHAR.
```

For information on this procedure in context, see the "Setting Message Properties" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setText

In Message Objects

This procedure clears the message body and sets a new text value. The call can be made when the message is in write-only or read-only mode. After the call, the message is in write-only mode, and additional appendText calls can be made to append more text:

SYNTAX

```
PROCEDURE setText.
DEFINE INPUT PARAMETER textValue AS CHAR.
```

For information on this procedure in context, see the "Setting Text" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setTransactedReceive

In Session Objects

This procedure makes the session transacted for receiving; a session is not transacted by default:

SYNTAX

```
PROCEDURE setTransactedReceive.
```

For information on this procedure in context, see the "Setting JMS Connection and Session Attributes" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setTransactedSend

In Session Objects

This procedure makes the session transacted for sending; a session is not transacted by default:

SYNTAX

```
PROCEDURE setTransactedSend.
```

For information on this procedure in context, see the "Setting JMS Connection and Session Attributes" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

setUser

In Session Objects

This procedure sets the user value for the SonicMQ broker login and overwrites the default user property set on the SonicMQ Adapter side:

SYNTAX

```
PROCEDURE setUser.
DEFINE INPUT PARAMETER User AS CHAR.
```

For information on this procedure in context, see the "Setting JMS Connection and Session Attributes" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

startReceiveMessages

In Session Objects

This procedure starts receiving messages after creating a new session or after calling stopReceiveMessages. Messages can be sent without calling startReceiveMessages:

SYNTAX

```
PROCEDURE startReceiveMessages.
```

NOTE: This procedure executes remotely (sends a message to the SonicMQ Adapter).

For information on this procedure in context, see the "Session-level Methods" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

stopReceiveMessages

In Session Objects

This procedure causes the SonicMQ Adapter broker to stop receiving messages on behalf of the 4GL client and to stop sending messages already received by the SonicMQ Adapter broker for the 4GL client. A subsequent call to startReceiveMessages resumes message reception and delivery.

If this procedure is called in a pubsubsession object and the subscription is not durable, messages published while reception is stopped are not delivered. A single message that was already sent to the client before stopReceiveMessages was called might be received by the client after the stopReceiveMessages call:

SYNTAX

```
PROCEDURE stopReceiveMessages.
```

NOTE: This procedure executes remotely (sends a message to the SonicMQ Adapter).

For information on this procedure in context, see the "Session-level Methods" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

subscribe

In Session Objects

This procedure subscribes to topicName. The messages are handled asynchronously by the messageConsumer object. A subscriptionName parameter with a value other than UNKNOWN specifies a durable subscription. Durable subscriptions require the JMS client to have a clientID identifier. The client must call setClientID in the pubsubsession object (or set the default clientID on the server side) if a durable subscription is desired. If the subscriptionName value is UNKNOWN or an empty string, the subscription is not durable. The default of noLocalPublications is false. The session, by default, get its own publications:

SYNTAX

```
PROCEDURE subscribe.
DEFINE INPUT PARAMETER topicName AS CHAR.
DEFINE INPUT PARAMETER subscriptionName AS CHAR.
DEFINE INPUT PARAMETER messageSelector AS CHAR.
DEFINE INPUT PARAMETER noLocalPublications AS LOGICAL.
DEFINE INPUT PARAMETER messageConsumer AS HANDLE.
```

NOTE: This procedure executes remotely (sends a message to the SonicMQ Adapter).

For information on this procedure in context, see the "Subscribing To a Topic" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

waitForMessages

In Session Objects

This procedure waits and processes events as long as the user-defined function, UDFName (in procH), returns true and there is no period of more than timeOut seconds in which no messages are received. The user-defined function is evaluated each time after a message is handled:

SYNTAX

```
PROCEDURE waitForMessages:
DEFINE INPUT PARAMETER UDFName AS CHAR NO-UNDO.
DEFINE INPUT PARAMETER procH AS HANDLE NO-UNDO.
DEFINE INPUT PARAMETER timeOut AS INT NO-UNDO.
```

For information on this procedure in context, see the "Processing Messages" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

writeBoolean

In Message Objects

This procedure writes boolean data to the body of a StreamMessage. An UNKNOWN value is considered false:

SYNTAX

```
PROCEDURE writeBoolean.
DEFINE INPUT PARAMETER value AS LOGICAL.
```

For information on this procedure in context, see the "Writing Data To a StreamMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

writeByte

In Message Objects

This procedure writes byte data to the body of a StreamMessage; byte values are –128 to 127. The server returns a NumberFormatException message for a value overflow:

SYNTAX

```
PROCEDURE writeByte.
DEFINE INPUT PARAMETER value AS INT.
```

For information on this procedure in context, see the "Writing Data To a StreamMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

writeBytesFromRaw

In Message Objects

This procedure writes bytes data to the body of a StreamMessage.

This procedure can be called in write-only mode to write additional bytes segments to a BytesMessage and work around the RAW data type limit of 32K:

SYNTAX

```
PROCEDURE writeBytesFromRaw.
DEFINE INPUT PARAMETER bytesValue AS RAW.
```

For information on this procedure in context, see the "Writing Data To a StreamMessage" and "Overcoming the 32K RAW Bytes Limit" sections in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

writeChar

In Message Objects

This procedure writes char data to the body of a StreamMessage; the number of characters in the char value must be one:

SYNTAX

```
PROCEDURE writeChar.
DEFINE INPUT PARAMETER value AS CHAR.
```

For information on this procedure in context, see the "Writing Data To a StreamMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

writeDouble

In Message Objects

This procedure writes double data to the body of a StreamMessage:

SYNTAX

```
PROCEDURE writeDouble.
DEFINE INPUT PARAMETER value AS DECIMAL.
```

For information on this procedure in context, see the "Writing Data To a StreamMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

writeFloat

In Message Objects

This procedure writes float data to the body of a StreamMessage:

SYNTAX

```
PROCEDURE writeFloat.
DEFINE INPUT PARAMETER value AS DECIMAL.
```

For information on this procedure in context, see the "Writing Data To a StreamMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

writeInt

In Message Objects

This procedure writes JMS int data to the body of a StreamMessage:

SYNTAX

```
PROCEDURE writeInt.
DEFINE INPUT PARAMETER value AS INT.
```

For information on this procedure in context, see the "Writing Data To a StreamMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

writeLong

In Message Objects

This procedure writes long data to the body of a StreamMessage. The fractional part of the DECIMAL value is truncated:

SYNTAX

```
PROCEDURE writeLong.
DEFINE INPUT PARAMETER value AS DECIMAL.
```

For information on this procedure in context, see the "Writing Data To a StreamMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

writeShort

In Message Objects

This procedure writes short data to the body of a StreamMessage. The server returns a NumberFormatException message for a value overflow:

SYNTAX

```
PROCEDURE writeShort.
DEFINE INPUT PARAMETER value AS INT.
```

For information on this procedure in context, see the "Writing Data To a StreamMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

writeString

In Message Objects

This procedure writes String data to the body of a StreamMessage:

SYNTAX

```
PROCEDURE writeString.
DEFINE INPUT PARAMETER value AS CHAR.
```

For information on this procedure in context, see the "Writing Data To a StreamMessage" section in Chapter 13, "Accessing SonicMQ Messaging From the Progress 4GL."

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X