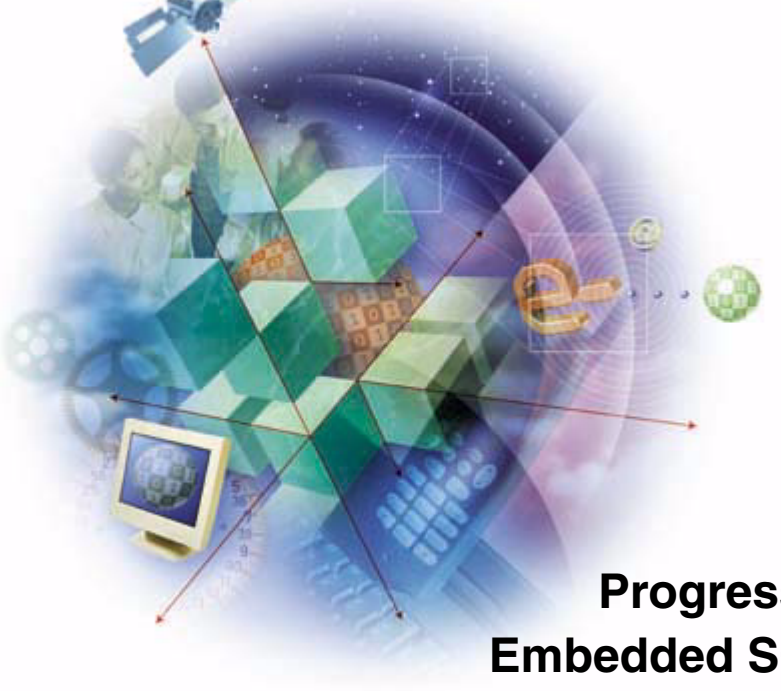

Progress Embedded SQL-89 Guide and Reference

© 2001 Progress Software Corporation. All rights reserved.

Progress® software products are copyrighted and all rights are reserved by Progress Software Corporation. This manual is also copyrighted and all rights are reserved. This manual may not, in whole or in part, be copied, photocopied, translated, or reduced to any electronic medium or machine-readable form without prior consent, in writing, from Progress Software Corporation.

The information in this manual is subject to change without notice, and Progress Software Corporation assumes no responsibility for any errors that may appear in this document.

The references in this manual to specific platforms supported are subject to change.

Progress, Progress Results, Provision and WebSpeed are registered trademarks of Progress Software Corporation in the United States and other countries. Apptivity, AppServer, ProVision Plus, SmartObjects, IntelliStream, and other Progress product names are trademarks of Progress Software Corporation.

SonicMQ is a trademark of Sonic Software Corporation in the United States and other countries.

Progress Software Corporation acknowledges the use of Raster Imaging Technology copyrighted by Snowbound Software 1993-1997 and the IBM XML Parser for Java Edition.

© IBM Corporation 1998-1999. All rights reserved. U.S. Government Users Restricted Rights — Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Progress is a registered trademark of Progress Software Corporation and is used by IBM Corporation in the mark Progress/400 under license. Progress/400 AND 400® are trademarks of IBM Corporation and are used by Progress Software Corporation under license.

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Any other trademarks and/or service marks contained herein are the property of their respective owners.

.

May 2001



Product Code: 4520
Item Number: 81088;9.1C

Contents

Preface . . . . . **xi**
Purpose xi
Audience xi
Organization of This Manual xi
Typographical Conventions xii
Syntax Notation xiii
Progress Messages xvii
Other Useful Documentation xix
Getting Started xix
Development Tools xx
Reporting Tools xxi
4GL xxii
Database xxiii
DataServers xxiii
SQL-89/Open Access xxiii
SQL-92 xxiv
Deployment xxiv
WebSpeed xxv
Reference xxv
SQL-92 Reference (These are non-Progress resources available from your technical bookseller.) xxvi

1. Introduction **1–1**
1.1 Progress/ESQL 1–2
1.1.1 Static and Dynamic ESQL 1–2
1.2 Prerequisites for Using ESQL 1–2
1.3 Progress/SQL and Progress/ESQL 1–3
1.3.1 Identifiers 1–4
1.3.2 Progress Function Support 1–4

1.4 Progress/ESQL Architecture 1–5
1.4.1 Large Client Model 1–5
1.4.2 Small Client Model 1–5
1.4.3 Small Client Model Versus Large Client Model 1–6
1.4.4 Configuration and Deployment 1–7
1.5 ESQL Directories and Files 1–7
1.6 Overview of Using ESQL 1–9
1.6.1 Designing and Programming the Application 1–10
1.6.2 Preprocessing the ESQL Source Files 1–11
1.6.3 Compiling the Preprocessed Source Files 1–14
1.6.4 Linking the ESQL Executable 1–14

2. Programming with ESQL 2–1
2.1 Programming Capabilities and Options 2–2
2.1.1 ESQL Sessions 2–2
2.1.2 Static and Dynamic ESQL 2–2
2.1.3 Progress/ESQL and ESQL-LIB 2–2
2.2 Writing an ESQL Application 2–3
2.3 Elements of Progress/ESQL 2–9
2.3.1 Basic Progress/ESQL Syntax 2–9
2.3.2 Reserved Words 2–9
2.3.3 Application Function Names 2–9
2.3.4 SQL Operators 2–10
2.3.5 Progress Function Support 2–12
2.3.6 3GL Access to Database Schema Triggers 2–13
2.3.7 ESQL Data Types 2–15
2.3.8 WHENEVER Statement 2–18
2.3.9 Progress/ESQL DDL Statements 2–20
2.4 Host Language Variables 2–21
2.4.1 Host Language Variables and ESQL-LIB 2–21
2.4.2 Host Language Variables and Dynamic ESQL 2–21
2.4.3 ESQL Standard Type Definitions and Conversions 2–22
2.4.4 Using Host Language Variables 2–23
2.5 Indicator Variables 2–24
2.5.1 Declaring and Specifying Indicator Variables 2–25
2.5.2 Using Indicator Variables to Exchange NULL Values 2–25
2.5.3 Using Indicator Variables to Retrieve NULL Values 2–26
2.5.4 Using Indicator Variables to Set NULL Values 2–26
2.6 Error Handling 2–26
2.6.1 SQLCODE Variable 2–27
2.6.2 SQLSTATE Variable 2–29
2.6.3 Progress Error Messages 2–33
2.6.4 Function Return Codes 2–33
2.7 Starting and Ending an ESQL Session 2–34

2.7.1 Starting a Session 2–34
2.7.2 Ending a Session 2–37
2.8 Connecting and Disconnecting Databases 2–37
2.8.1 Connecting Databases 2–38
2.8.2 Disconnecting Databases 2–39
2.9 Managing Transactions in ESQL 2–40
2.9.1 Committing Work 2–41
2.9.2 Rolling Back Work 2–41
2.9.3 Using DDL Statements in Transactions 2–41
2.10 Checking ESQL Run-time Privileges 2–41
2.10.1 Using the CAN-DO Function in a Singleton SELECT 2–42
2.10.2 Using the CAN-DO Function in a WHERE Clause 2–44
2.11 Managing Multi-user Applications 2–44
2.11.1 Application Lock Management 2–46
2.12 Using Dynamic Embedded SQL 2–47
2.12.1 Differences Between Dynamic and Static ESQL 2–47
2.13 Executing Dynamic ESQL 2–49
2.13.1 Immediate Execution 2–49
2.13.2 Prepared Execution 2–49
2.13.3 Elements of Dynamic ESQL Execution 2–50
2.14 Using Immediate Execution 2–51
2.15 Using Prepared Execution 2–52
2.15.1 Elements of Prepared Execution 2–53
2.15.2 Data Exchange in Prepared Execution 2–53
2.15.3 Forms of Prepared Execution 2–54
2.15.4 Prepared Execution for Dynamic Cursor Specification 2–54
2.15.5 Prepared Execution for Non-cursor Specification 2–55
2.16 SQL Descriptor Area 2–56
2.16.1 Functions for Working with SQL Descriptor Areas 2–57
2.16.2 SQLDA_T Structure Components 2–61
2.16.3 SQLVAR_T Structure Components 2–62
2.16.4 SQLDA Data Type Management 2–64
2.17 Executing Dynamic Cursor Specifications 2–65
2.17.1 Dynamic Cursor SELECT Using Progress/ESQL 2–66
2.18 Using Dynamic Parameter Markers 2–70
2.18.1 Parameter Marker Syntax 2–70
2.18.2 Determining Parameter Marker Data Types 2–72
2.18.3 Writing an Application with Parameter Markers 2–73
2.18.4 Using Progress/ESQL and Parameter Markers 2–74
2.18.5 Using ESQL-LIB and Parameter Markers 2–76
2.19 Dynamic ESQL Statements for Prepared Execution 2–78
2.19.1 Dynamic PREPARE Statement 2–78
2.19.2 Dynamic ALLOCATE DESCRIPTOR Statement 2–79
2.19.3 Dynamic DESCRIBE Statement 2–80
2.19.4 Dynamic DECLARE CURSOR Statement 2–80

2.19.5 Dynamic OPEN Statement 2–81
2.19.6 Dynamic EXECUTE Statement 2–82
2.19.7 Dynamic FETCH Statement 2–84
2.19.8 Dynamic CLOSE Cursor Statement 2–86
2.19.9 Dynamic DEALLOCATE DESCRIPTOR Statement 2–86

3. Building and Running ESQL Applications 3–1
3.1 Choosing an ESQL Configuration 3–2
3.1.1 ESQL Large Client Model 3–2
3.1.2 ESQL Small Client Model 3–3
3.1.3 Differences Between Small and Large Clients 3–6
3.2 Overview of Building an ESQL Executable 3–6
3.3 Preprocessing Progress/ESQL Source Files 3–7
3.3.1 Precompile Mode 3–8
3.3.2 Non-precompile Mode 3–9
3.3.3 ESQL Preprocessor Syntax and Operation 3–10
3.4 Compiling ESQL Source Files 3–12
3.4.1 Compiling on UNIX Systems 3–12
3.4.2 Compiling in Windows 3–13
3.5 Building Applications Containing Windows Resources 3–13
3.6 Linking ESQL Executables 3–14
3.6.1 ESQL Client Executables 3–14
3.6.2 Linking for Windows with ESQL Small Client 3–14
3.6.3 PROBUILD and the Linker 3–16
3.7 Running ESQL Applications 3–17
3.7.1 Preparing to Run Small and Large Client Applications 3–17
3.7.2 Starting ESQL Small Clients 3–20
3.7.3 Connecting Small Clients to Open Interface Drivers 3–21
3.7.4 Setting OIB and OID UNIX Environment Variables 3–22
3.7.5 Starting UNIX Open Interface Brokers 3–23
3.7.6 Starting an OID Directly Without a Broker 3–24
3.7.7 Starting the Small Client Processes 3–24
3.7.8 Starting Local Small Clients 3–26
3.7.9 Organizing OID and Database Startup Parameters 3–26
3.7.10 Shutting Down an ESQL Small Client Application 3–27
3.8 The ESQL Demo Applications 3–27
3.8.1 Static Embedded SQL Demo Applications 3–27
3.8.2 Dynamic Embedded SQL Demo Applications 3–29

4. ESQL-LIB Reference 4–1
4.1 Overview 4–2
4.1.1 Error and Status Returns 4–2
4.1.2 SQL Request Handles 4–2
4.1.3 Static ESQL Variables and Literal Constants 4–3

4.1.4 Dynamic ESQL and the SQL Descriptor Area 4–3
4.1.5 ESQL-LIB Functions 4–4
4.1.6 General Diagnostics. 4–7
4.2 ESQL-LIB Reference Entries 4–9
sqlald() — Allocate Dynamic SQL Descriptor 4–10
sqlalloc() — Allocate Dynamic Memory 4–11
sqlapplname() — Set the ESQL Application Name 4–13
sqlcdbind() — Bind Your Error Code Address to ESQL 4–15
sqlcommit() — Commit the Database Transaction 4–17
sqlconn() — Connect Database(s) 4–19
sqldelrow() — Delete Row at Open Cursor 4–20
sqldinit() — Initialize an Allocated SQL Descriptor Area 4–22
sqldiscon() — Disconnect a Database 4–24
sqldld() — Deallocate Dynamic Descriptor Area 4–25
sqldsize() — Calculate the Size of an SQL Descriptor Area 4–27
sqldynclose() — Close an SQL Statement 4–29
sqldyndesc() — Get Dynamic Column Descriptions 4–32
sqldyndescinp() — Get Dynamic Parameter Descriptions 4–34
sqldynexec() — Execute a Dynamic SQL Statement 4–36
sqldynftch() — Fetch Next Row of Dynamic Cursor 4–38
sqldyngetn() — Get a Cursor Name 4–40
sqldynncols() — Get the Number of Columns in a Result Set 4–42
sqldynnparms() — Get the Number of Parameter Markers 4–44
sqldynposfetch() — Fetch Row of Dynamic Open Cursor 4–46
sqldynprep() — Prepare a Dynamic SQL Statement 4–49
sqldynrq() — Execute a Dynamic Non-SELECT Statement 4–52
sqldynsetn() — Define a Cursor Name 4–54
sqlfetch() — Fetch Next Row of Open Cursor 4–56
sqlfree() — Deallocate Dynamic Memory 4–58
sqlgetmsg() — Get Next Error Message 4–60
sqlgetrqhdl() — Get a Dynamic Request Handle 4–62
sqllogin() — Start a Progress ESQL Session 4–64
sqllogout() — Terminate a Progress ESQL Session 4–67
sqlrequest() — Execute a Static ESQL Statement 4–68
sqlrollback() — Rollback Any Pending Transaction 4–71
sqlsetlogoption() — Set a Session-wide Option 4–72
sqlupdrow() — Update Row at Open Cursor 4–78

A. Progress ESQL and ANSI Standards A–1
A.1 Data Type Conversion Rules A–2
A.1.1 Assignment Rules A–2
A.1.2 Comparison Rules A–2
A.1.3 UNION Data Type Compatibility A–3
A.2 Progress/ESQL Not Addressed in 1989 SQL Standard A–3

A.3 Standard Behavior Available Only with -Q or -Q2 . . . . . A–4
A.4 Progress/ESQL Extensions to 1989 SQL A–4

B. ESQL Data Type Compatibilities B–1
B.1 SQL and Core Progress Data Types B–2
B.2 ESQL Standard C Data Type Definitions B–3
B.3 Static ESQL Data Type Matching B–3
B.4 Dynamic ESQL Data Type Matching and Conversion B–4
B.4.1 The SQL Descriptor Area and Data Type Conversion B–5
B.4.2 Output Data Conversion B–8
B.4.3 Input Data Conversion B–14

C. V6 HLI to V9 ESQL Migration C–1
C.1 Migrating SQL Preprocessor-based Applications C–2
C.1.1 Using Dynamic ESQL C–2
C.1.2 Calling sqllogout() C–2
C.1.3 Using Host Language Variables C–2
C.1.4 Preparing the Executable C–3
C.2 Migrating ESQL-LIB-based Applications C–3
C.2.1 Changing Calls to sqlcdbind() C–3
C.2.2 Changing sqlv[] Array References C–3
C.2.3 Changing Function Return Behavior C–3
C.2.4 Replacing Calls to sqldynopen() C–4
C.2.5 Replacing Calls to sqlclose() C–4
C.2.6 Removing Obsolete Functions C–4
C.2.7 Preparing the Executable C–4

Index Index-1

Figures

Figure 1–1: Large Client Model 1–5
Figure 1–2: Small Client Model 1–6
Figure 1–3: Steps to Build an ESQL Executable 1–9
Figure 1–4: ESQL Sample Source File Fragment 1–10
Figure 1–5: Preprocessed Source File Fragment (Not Precompiled) 1–12
Figure 1–6: Preprocessed Source File Fragment (Precompiled) 1–13
Figure 1–7: Large Client with Compiler-functional Layers 1–15
Figure 2–1: Sample ESQL Application (Not Preprocessed) 2–4
Figure 2–2: . . . . . Sample ESQL Application (Preprocessed) 2–8
Figure 2–3: SQLDA_T Memory Map After Allocation 2–60
Figure 3–1: Large Client Process Model 3–2
Figure 3–2: Typical Local Large Client Configurations 3–3
Figure 3–3: Small Client Process Model 3–4
Figure 3–4: Typical Remote Small Client Configuration 3–5
Figure 3–5: Sample Linker Response File 3–15

Tables

Table 1–1: Default ESQL Directory 1–7
Table 1–2: Windows-only Files in Default ESQL Directory 1–8
Table 2–1: Progress/SQL Operators 2–10
Table 2–2: Progress Function Support in ESQL 2–12
Table 2–3: Invalid 4GL Keywords in ESQL Schema Triggers 2–14
Table 2–4: Parameter Values of sqlcdbind() 2–19
Table 2–5: SQLCODE Return Code Values 2–27
Table 2–6: Meanings of the +100 Return Code 2–27
Table 2–7: SQLCODE Error Code Values 2–28
Table 2–8: SQLSTATE Error Code Values 2–30
Table 2–9: Progress Record Locking Phrases 2–44
Table 2–10: SQL Statements and Associated Record Locks 2–45
Table 2–11: ESQL-LIB Routines 2–57
Table 2–12: SQLDA_T Structure Components 2–61
Table 2–13: SQLVAR_T Structure Components 2–62
Table 2–14: Default Parameter Marker Data Types 2–72
Table 2–15: Data Types for Target Storage 2–85
Table 3–1: Common ESQL Startup Parameters 3–18
Table 3–2: OID Connection (-SV) Parameter List 3–21
Table 3–3: OIB and OID UNIX Environment Variables 3–22
Table 4–1: Static ESQL Variable Declarations 4–3
Table 4–2: ESQL-LIB Functions 4–4
Table 4–3: Normalized Function Returns 4–7
Table 4–4: Exceptional Function Returns 4–8
Table 4–5: Lock Callback Function Return Codes 4–73
Table B–1: SQL and Progress Core Data Types B–2
Table B–2: Progress/SQL Data Types and C Equivalents B–3
Table B–3: Matching Dynamic SQL and C Data Types B–5
Table B–4: Precision of SQL Descriptor Data Types B–6
Table B–5: Scale of SQL Descriptor Data Types B–6
Table B–6: Output Length of SQL Descriptor Data Types B–7
Table B–7: Dynamic Output from a CHARACTER Column B–8
Table B–8: Dynamic Output from an INTEGER Column B–10
Table B–9: Dynamic Output from a DECIMAL Column B–11
Table B–10: Dynamic Output from a DATE Column B–12
Table B–11: Dynamic Output from a LOGICAL Column B–13
Table B–12: Dynamic Input from SQL Character Data Types B–14
Table B–13: Dynamic Input from SQL Date Data Types B–15
Table B–14: Dynamic Input from SQL Decimal Data Types B–16
Table B–15: Dynamic Input from SQL Floating Point Data Types B–17
Table B–16: Dynamic Input from SQL Integer Data Types B–17
Table B–17: Dynamic Input from SQL Logical Data Types B–18

Preface

Purpose

You can easily build many database applications using the Progress 4GL. If necessary, you can use Progress/SQL within a 4GL procedure to perform SQL operations such as joins. However, there are many instances where you might need to build an application in a host language, such as C, that can also access Progress databases and DataServers. For these applications, you can use Progress Embedded SQL (ESQL). Progress ESQL provides an industry-standard means for your application to read and write Progress data while running in a variety of software configurations and environments.

This manual describes ESQL and the steps required to use it in your C applications. It also provides information on the ANSI standards with which ESQL complies and describes the available Progress extensions that you can use to take advantage of special Progress data handling capabilities. This is not a tutorial on Structured Query Language (SQL).

Audience

This manual is intended for any programmer or software engineer experienced in building database applications using the C language, and who is thoroughly familiar with general database programming concepts and with the use of SQL.

Organization of This Manual

Chapter 1, “Introduction”

Describes the elements of ESQL, including its programming tools and deployment architecture, and provides an overview of the steps required to build ESQL application.

Chapter 2, “Programming with ESQL”

Describes how to produce ESQL source files, using ESQL programming tools to specify SQL statements within your C applications.

Chapter 3, “Building and Running ESQL Applications”

Describes how to choose among the deployment options available with ESQL architecture, and how to turn your ESQL source files into ESQL executables.

Chapter 4, “ESQL-LIB Reference”

Describes the ESQL-LIB interface, including the data definitions and C function calls that support ESQL application programming.

Appendix A, “Progress ESQL and ANSI Standards”

Describes where Progress/ESQL complies with and extends the ANSI standard upon which ESQL is based.

Appendix B, “ESQL Data Type Compatibilities”

Describes how SQL data types are supported by ESQL, including the large variety of data conversions between C and SQL.

Appendix C, “V6 HLI to V9 ESQL Migration”

Describes the changes you make to a Version 6 HLI application to run it as a Version 9 ESQL application.

Index

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 ... ] }
       }
```

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)
```

Use Progress online help to get more information about Progress messages. On the Windows platform, 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, you can 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 guide 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.

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 commands and parameters in alphabetical order.

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 External Program Interfaces

A guide to accessing non-Progress applications from Progress. This guide describes how to use system clipboards, UNIX named pipes, Windows dynamic link libraries, Windows dynamic data exchange, Windows ActiveX controls, and the Progress Host Language Call Interface to communicate with non-Progress applications and extend Progress functionality.

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 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, for example, the *Progress DataServer for ODBC Guide*, the *Progress DataServer for ORACLE Guide*, or the *Progress/400 Product 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 3.1 or higher 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/prodindex/acrobat/readstep.html.

SQL-89/Open Access

Progress Open Client Developer's Guide

A guide that describes how to write and deploy Java and ActiveX applications that run as clients of the Progress AppServer. The guide includes information about how to expose the AppServer as a set of Java classes or as an ActiveX server.

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 Version 3 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.

SQL-92 Reference (These are non-Progress resources available from your technical bookseller.)

A Guide to the SQL Standard

Date, C.J., with Hugh Darwen. 1997. Reading, MA: Addison Wesley.

Understanding the New SQL: A Complete Guide

Melton, Jim (Digital Equipment Corporation) and Alan R. Simon. 1993. San Francisco: Morgan Kaufmann Publishers.

1

Introduction

Progress Embedded Structured Query Language (ESQL) allows you to write C applications that access Progress through a standards-based ESQL interface. This interface consists of two major components:

- **Progress/ESQL** — A collection of ESQL statements that allow you to specify and direct the execution of Progress/SQL statements from within a C source file. The SQL preprocessor converts these statements into C source code.
- **ESQL-LIB** — A collection of C functions and data structures that support Progress/ESQL syntax and the management of ESQL sessions at run time.

Starting with a minimum set of session management functions from ESQL-LIB, you can code your ESQL application using either Progress/ESQL, ESQL-LIB, or both, depending on the type of ESQL you are using.

Progress/ESQL conforms to ANSI standard X3.135-1989 Level 2, as verified by NIST Test Suite Version 3.0. ESQL is supported on both 32 bit and 64 bit platforms.

This chapter describes:

- Progress/ESQL features and architecture
- Files provided with your ESQL installation
- Using ESQL
- An overview of programming and building an ESQL application

1.1 Progress/ESQL

Progress/ESQL allows you to embed SQL statements within a program written in a host language. A host language allows you to build applications that compile and run in the native machine language of the application platform. The host language Progress/ESQL supports is C. Using ESQL, you can use:

- Your host language to handle your application's user interface, flow of control, and access to non-Progress features and capabilities
- Progress features that safeguard your data with crash-proof data recovery, including online transaction processing, and two-phase commit when updating more than one Progress database in a single transaction
- Progress DataServer access to many popular data managers, including the ability to perform SQL join operations across DataServers
- Many other features of the Progress Application Development Environment (ADE)

1.1.1 Static and Dynamic ESQL

Progress/ESQL supports two basic types of Embedded SQL:

- **Static Embedded SQL** — SQL statements fully specified at host language compile time, with the option to precompile the SQL syntax before application run time. Precompiling your SQL can save significant execution time and space on some platforms.
- **Dynamic Embedded SQL** — SQL statements fully specified at application run time, when they are compiled and executed dynamically. With dynamic ESQL, you can specify the exact form of your SQL statements based on run-time input. Dynamic ESQL also features dynamic parameter markers, which are place holders for values that you supply in run-time SQL expressions. Using parameter markers, you can execute the same SQL statement repeatedly, substituting different values for the parameters each time.

1.2 Prerequisites for Using ESQL

To use ESQL, you should be familiar with the following topics:

- The Progress 4GL/RDBMS. If you are new to Progress, see the *Progress Language Tutorial for Windows* or *Progress Language Tutorial for Character*
- Progress/SQL programming. For information, see the *Progress SQL-89 Guide and Reference*.

- C language programming. For information, see the many references and guides available in the public domain and from C language vendors.
- Progress/ESQL and ESQL-LIB functions to access databases from a C application. For more information, see Chapter 2, "Programming with ESQL."
- The SQL preprocessor to translate an ESQL source file into a C application module that you can compile and link into an ESQL application. For more information, see Chapter 3, "Building and Running ESQL Applications."
- The Progress End User Configuration (`PROBUILD`) utility. For information on `PROBUILD`, see the *Progress Client Deployment Guide*.

1.3 Progress/SQL and Progress/ESQL

Progress/ESQL allows you to access most Progress/SQL syntax. This includes all ANSI standard elements, but excludes some Progress/SQL extensions to the ANSI standard. Progress/ESQL also includes additional elements to specify SQL requests from a C program according to the ANSI standard. For more information on Progress/SQL, see the *Progress SQL-89 Guide and Reference*. For ESQL standard information, see a copy of the *ANSI Standard X3.135-1989 Level 2*. For information on how Progress/ESQL supports and extends the ANSI standard, see Appendix A, "Progress ESQL and ANSI Standards."

Progress/ESQL is a mechanism to specify SQL statements directly in your host language program. SQL statements in Progress/SQL procedures are called interactive SQL. You enter procedures that contain Progress statements using the Procedure Editor .Progress/ESQL provides an alternative to tools such as the Procedure Editor, for specifying Progress/SQL statements to Progress.

When you enter and execute Progress/SQL from the Procedure Editor, you are using a form of SQL that integrates well with Progress 4GL procedures. When you embed Progress/SQL in a host language program, you use Progress/ESQL (and ESQL-LIB) to provide a bridge between the host language environment and the Progress environment. Progress/ESQL uses a form of Progress/SQL similar to the form in the Progress 4GL.

When you specify a Progress/SQL statement using static ESQL, you replace the Progress statement period terminator (.) with the C language semicolon terminator (;). Also, wherever a Progress variable appears in Progress/SQL, you must use an ESQL-declared host language variable prefixed with a colon (:) in Progress/ESQL.

Certain Progress extensions to SQL are not available in Progress/ESQL. For example, you cannot specify a SELECT statement using the WITH option to access Progress I/O functionality. All SELECT statements in ESQL must either be cursor SELECT statements or singleton SELECT statements that return values to host language variables.

You can reference columns and host language variables only by name. You cannot use most Format Phrase options or array subrange expressions (*variable* [*expression* [FOR *n*]]) that apply to columns and variables in Progress/SQL. However, you can use the COLUMN-LABEL, LABEL, and FORMAT options, especially in the CREATE TABLE statement. You can also qualify both table and column names for multiple schemas (databases).

For more information on Progress SQL, see the *Progress SQL-89 Guide and Reference*.

1.3.1 Identifiers

When you reference Progress identifiers (for example, table fields (columns) or function names) that contain hyphens in Progress/ESQL, precede the hyphen with a backslash (\) to escape the hyphen. If you do not insert a backslash before the hyphen, the SQL preprocessor interprets the hyphen (-) as a minus sign because the hyphens permitted in Progress are not permitted in ANSI standard SQL identifiers. In the following example, a backslash is inserted in the column name cust-num so that the SQL preprocessor interprets it correctly:

```
WHERE cust\-num < 1000;
```

NOTE: You can reference Progress-hyphenated names and language elements without escaping the hyphens by starting both the SQL preprocessor and your application using the ESQL No Padding (-esqlnopad) startup parameter. For more information, see Chapter 3, "Building and Running ESQL Applications."

When including Progress identifiers in a string in C, use double backslashes (\\ ) to escape the hyphen:

```
WHERE cust\\-num < 1000;
```

1.3.2 Progress Function Support

Some Progress functions are supported in ESQL. You can use these Progress functions in ESQL, but all other Progress functions permitted in Progress/SQL are not permitted in ESQL. ANSI standard SQL aggregate functions (such as AVG and COUNT) are permitted in Progress/ESQL. For a complete listing of the functions supported in ESQL, see "Progress Function Support" in Chapter 2, "Programming with ESQL."

When you reference Progress function names that contain hyphens in Progress/ESQL, precede the hyphen with a backslash (\) to escape the hyphen. See the "Identifiers" section for more information.

1.4 Progress/ESQL Architecture

Progress allows you to build an ESQL application using two distinct configurations—a large client model or a small client model.

1.4.1 Large Client Model

If you use the large client model, your ESQL application contains a complete Progress client executable (local or remote), where your ESQL application module replaces the Progress ADE tools, as shown in Figure 1–1. Using this model, your ESQL application feeds SQL requests (statements to execute) directly to the Progress run-time Compiler and Interpreter within the same large client executable.

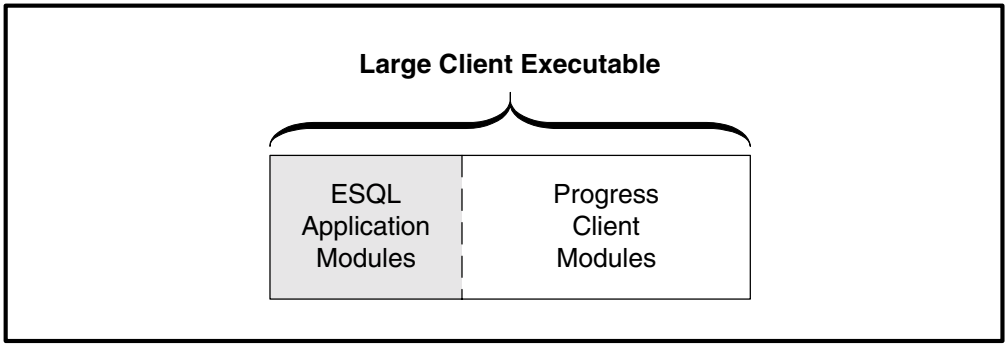

Figure 1–1: Large Client Model

1.4.2 Small Client Model

If you use the small client model, your ESQL application is separated into two distinct executables that communicate through Progress local interprocess communications (IPC) or network communications (NC) modules as shown in Figure 1–2. These two executables are:

- The small client executable that contains all of your ESQL application code and Progress client communication modules
- An *Open Interface Driver* (OID) that interprets SQL requests from an ESQL small client application

  The OID is essentially a Progress database client process that receives SQL input from the small client, feeds it directly to the Progress run-time Compiler and Interpreter, and returns the results to the small client.

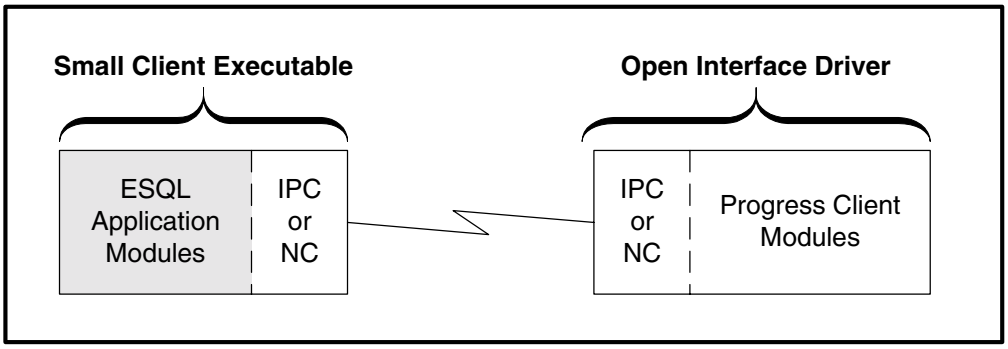

Figure 1–2: Small Client Model

The Open Interface Architecture and the Small Client Model

Progress supports the ESQL small client model as part of its Open Interface Architecture. This architecture enables non-Progress applications to access Progress using several standard data interfaces, including SQL. During execution, an Open Interface Broker actually starts and assigns an OID for each ESQL small client application that accesses Progress.

1.4.3 Small Client Model Versus Large Client Model

Separating your ESQL large client application into separate small client and OID processes provides the following advantages:

- Application Isolation

  The OID effectively isolates all Progress database client code, including signals and global references, from your ESQL application environment. This enables you to:

  – Use signals as your application requires.

  – Open your global name space, thereby reducing the potential for name conflicts with your application code.

  – Significantly reduce the size of your application attributable to Progress-specific code.

- Client Maintainability

  An ESQL small client is much smaller and easier to maintain, especially on smaller workstations. You need only link a small Progress communications layer with your application code.

- Client Upgradeability

  In many instances, your ESQL small client application can automatically benefit from enhancements to Progress database client/server technology without relinking your application code.

- Network Efficiency

  In a network configuration in which the database resides on a separate machine, the OID can provide more efficient operation by minimizing network traffic. It performs all of the SQL table manipulation and result set processing that any Progress database client provides, sending only the requested data back to your remote ESQL small client.

1.4.4 Configuration and Deployment

You can configure and deploy your ESQL application in a multi-user environment using one or both of the large and small client models. For more information on whether to use one or both of these models for your ESQL application, see Chapter 3, "Building and Running ESQL Applications."

1.5 ESQL Directories and Files

The default ESQL installation process creates the following directories:

- UNIX: `$DLC/probuild/esqlc`
- MS-Windows: `%DLC%\PROBUILD\ESQLC`

Table 1–1 lists the files in this directory.

Table 1–1: Default ESQL Directory *(1 of 2)*

File	Description
`proesql.h`	A C header file that contains definitions and declarations required to reference ESQL-LIB.
`esqldemo.cc`	A character mode ESQL sample program that demonstrates static Progress/ESQL. This program works on UNIX or MS-Windows. Its GUI mode counterpart is `wesqldmo.cc`.
`esqldm64.cc`	A 64-bit version of an ESQL sample program that demonstrates static Progress/ESQL.

Table 1–1: Default ESQL Directory *(2 of 2)*

File	Description
dyndemo.cc	An ESQL sample program that demonstrates dynamic Progress/ESQL.
clidemo.c	An ESQL sample program specified at the ESQL-LIB level that demonstrates dynamic ESQL using parameter markers.
dprmdemo.cc	An ESQL sample program that uses a combination of ESQL-LIB calls and ESQL statements to demonstrate dynamic ESQL using parameter markers.

Table 1–2 lists the Windows-only files in this directory.

Table 1–2: Windows-only Files in Default ESQL Directory

File	Description
wesqldmo.cc	A GUI mode ESQL sample program that demonstrates static Progress/ESQL. This program works on MS-Windows only. Its character mode counterpart is esqldemo.cc.
esql32.lib	32 bit ESQL library
prosql32.lib	32 bit ODBC library
wcesql.rc	Windows resource file for ESQL applications
wcesql.res	Windows binary resource file for ESQL applications
wcesql.h	Windows header file for ESQL applications.
wcesqlnt.def	Windows resource definition file for ESQL applications

The default ESQL directory also contains a subdirectory called bin which contains the SQL preprocessor. The SQL preprocessor is named sqlccp in UNIX and sqlcpp.exe in Windows.

1.6 Overview of Using ESQL

Figure 1–3 shows the basic steps to build an ESQL application. The shaded boxes represent the files that ultimately compose the application you deploy.

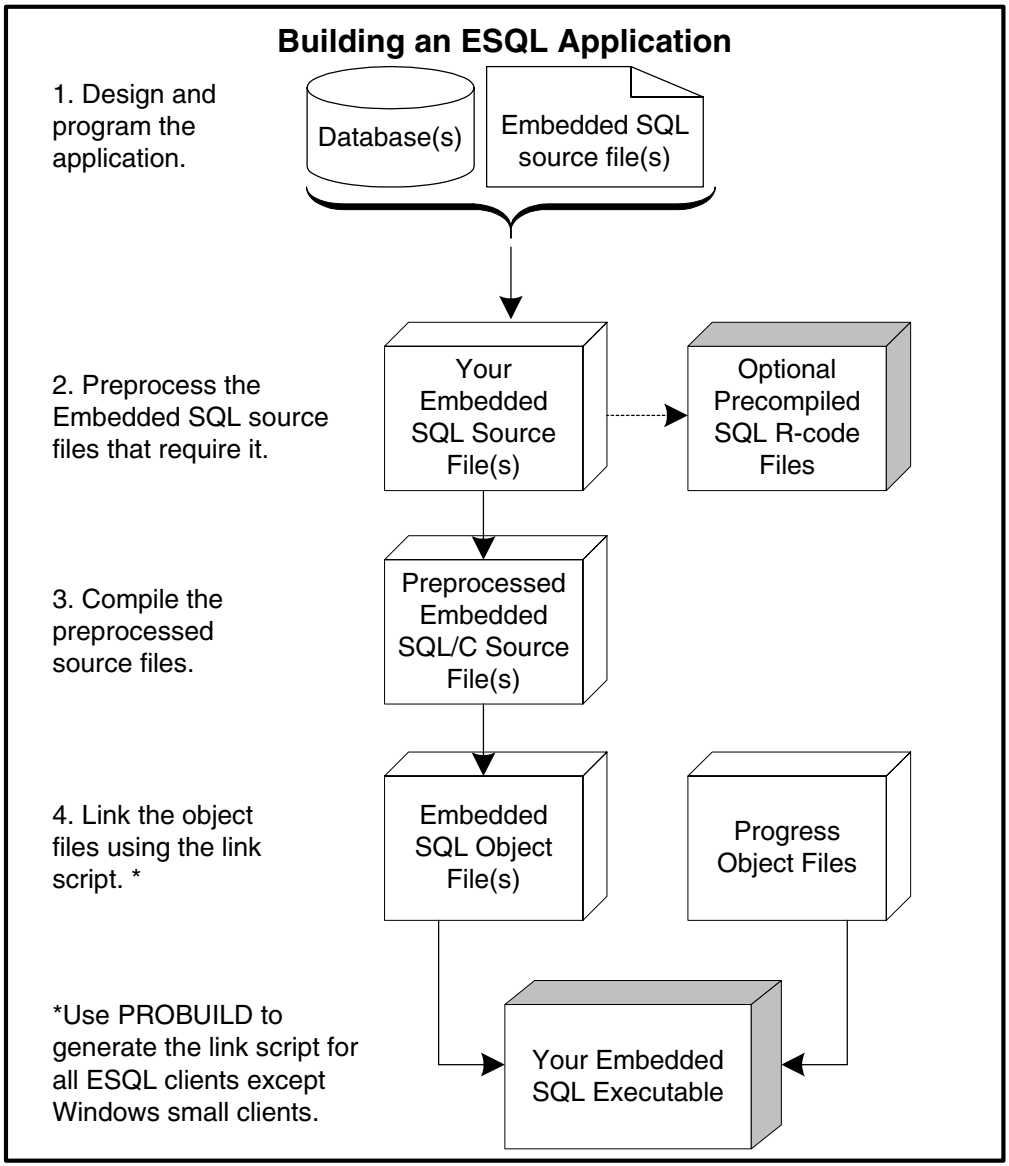

Figure 1–3: Steps to Build an ESQL Executable

1.6.1 Designing and Programming the Application

The first step in using Progress/ESQL is to design and program your ESQL application. An ESQL application consists of the database(s) you use to store data and one or more ESQL *source files*. An ESQL source file is a program written in C that contains ESQL statements and ESQL-LIB function calls.

Figure 1–4 shows a fragment of an ESQL source file that contains ESQL statements. The bold text shows the ESQL-LIB function calls.

```
while (1) /* loop until last customer record read */
{
     EXEC SQL fetch x into :custnum,
               :nam, :city, :state, :maxcred;
     printf("%-7d%-22s%-14s%-7s%8d\n", custnum, nam, city, state, maxcred);
     if (custnum > hicustnum - 3)
     {
         EXEC SQL delete from customer where current of x;
         printf("Record for customer %d deleted\n",custnum);
     }
}
                         .
                         .
                         .
```

Figure 1–4: ESQL Sample Source File Fragment

Consider the following options when designing your application:

- Can you define all SQL statements at compile time or do you need to allow for SQL requests known only at run-time? To specify requests at run time, use dynamic ESQL. Otherwise, static ESQL is sufficient.
- Do you need to specify repeated run-time SQL requests that differ only in the parameter values used for selection or input criteria? If so, you can implement dynamic ESQL calls with parameter marker capability.
- Does your application have specialized signal or memory requirements, and if so, does it run on a single platform or on a network? You can meet your requirements by planning for a small or large client environment. Your choice might affect how you specify your own functions and signal processing requirements (in a UNIX environment).

For more information on designing and coding your ESQL application, see Chapter 2, "Programming with ESQL."

1.6.2 Preprocessing the ESQL Source Files

After you write your ESQL source files, preprocess them using the SQL preprocessor (`sqlcpp`). The preprocessor creates a pure C source file for each ESQL source file that it processes.

NOTE: Complete this step only for ESQL source files that contain static and dynamic Progress/ESQL statements. Files that only contain calls to dynamic ESQL-LIB functions do not require preprocessing. However, you must ensure that any files you do not preprocess contain any necessary data declarations that the preprocessor provides.

As it generates output, the preprocessor:

- Inserts the #include C preprocessor directive to specify the `proesql.h` header file and any data declarations required for cursors or other identifiers specified in Progress/ESQL statements.
- Replaces ESQL statements with ESQL-LIB function calls and data references. The ESQL statements are not deleted, but commented out so you can see how each ESQL statement is converted to ESQL-LIB function calls and data assignments.
- Optionally inserts additional comments that contain readable representations of certain function parameter assignments, such as the static ESQL statements assigned to character arrays in generated function calls.
- Optionally precompiles any static ESQL statements in your application (except for Data Definition Language (DDL) statements). Precompilation generates a Progress r-code file for each SQL statement. R-code files contain the canonical (parsed and semantically structured) run code that Progress actually executes at run time for the specified SQL statements. If you do not precompile, Progress compiles the SQL statements when the application executes, at a cost in execution speed.

  If you precompile, run all necessary database servers and provide the command line database connection parameters that enable the preprocessor to connect to the database(s) that your application accesses at run time. The preprocessor acts as a Progress database client and invokes the Progress Compiler to compile your SQL statements with these databases.

Figure 1–5 and Figure 1–6 demonstrate the differences between C source files preprocessed without precompilation and with it. The bold code shows the ESQL-LIB function calls.

Preprocessing Without Precompilation

Figure 1–5 shows the fragment from Figure 1–4 after preprocessing without precompilation.

```
    while (1) /* loop until last customer record read */
    {
        /* EXEC SQL fetch x into :custnum,
                  :nam, :city, :state, :maxcred; */
        {
static char sqltxt[] = {'f','e','t','c','h',' ','x',' ','i','n','t','o','
','e','s','q','l','o','0',',',',','e','s','q','l','o','1',',',',','e','s','q'',
'l','o','2',',',',','e','s','q','l','o','3',',',',','e','s','q','l','o','4'','.
',0};
static char sqlfmt[] = {'4','i','3','0','s','2','0','s','3','s','4','i',0};
static char *sqlv[6];
            sqlv[0] = (char *) &custnum;
            sqlv[1] = (char *) nam;
            sqlv[2] = (char *) city;
            sqlv[3] = (char *) state;
            sqlv[4] = (char *) &maxcred;
            sqlv[5] = (char *) 0;
            sqlcdbind(&SQLCODE, (char *)0);
            sqlfetch(sql0,(int) 0x08,sqltxt,sqlfmt,sqlv);
            if (SQLCODE == 100) goto done1;
            if (SQLCODE < 0) goto err;
        }
       printf("%-7d%-22s%-14s%-7s%8d\n",custnum, nam, city, state, maxcred);
        if (custnum > hicustnum - 3)
        {
            /* EXEC SQL delete from customer where current of x; */
            {
                sqlcdbind(&SQLCODE, (char *)0);
                sqldelrow(sql0,(int)0x010,"");
                if (SQLCODE == 100) goto done1;
                if (SQLCODE < 0) goto err;
            }
            printf("Record for customer %d deleted\n", custnum);
        }
    }

.

.

.
```

Figure 1–5: Preprocessed Source File Fragment (Not Precompiled)

Without precompilation, each ESQL-LIB function is passed either a character string that contains the text of the corresponding SQL statement (sqltxt in the figure) or a null string, depending on the function.

Preprocessing with Precompilation

Figure 1–6 shows the fragment in Figure 1–4 after preprocessing with precompilation. The bold code shows the ESQL-LIB function calls.

```
    while (1) /* loop until last customer record read */
    {
        /* EXEC SQL fetch x into :custnum,
                  :nam, :city, :state, :maxcred; */
        {
static char sqlfmt[] = {'4','i','3','0','s','2','0','s','3','s','4','i',0};
static char *sqlv[6];
            sqlv[0] = (char *) &custnum;
            sqlv[1] = (char *) nam;
            sqlv[2] = (char *) city;
            sqlv[3] = (char *) state;
            sqlv[4] = (char *) &maxcred;
            sqlv[5] = (char *) 0;
            sqlcdbind(&SQLCODE, (char *)0);
            sqlfetch(sql0,(int) 0x0208,"OBCA34.p",sqlfmt,sqlv);
            if (SQLCODE == 100) goto done1;
            if (SQLCODE < 0) goto err;
        }
       printf("%-7d%-22s%-14s%-7s%8d\n",custnum, nam, city, state, maxcred);
        if (custnum > hicustnum - 3)
        {
            /* EXEC SQL delete from customer where current of x; */
            {
                sqlcdbind(&SQLCODE, (char *)0);
                sqldelrow(sql0,(int)0x0210,"AFAD67.p");
                if (SQLCODE == 100) goto done1;
                if (SQLCODE < 0) goto err;
            }
            printf("Record for customer %d deleted\n", custnum);
        }
    }

.

.

.
```

Figure 1–6: Preprocessed Source File Fragment (Precompiled)

With precompilation, the preprocessor passes the r-code filename created to execute the corresponding SQL statement to each ESQL-LIB function.

Working with Preprocessor Output

The preprocessor generates a C source file with the same name as the ESQL source file but with a different extension. (By convention, the ESQL source file extension is `.cc`, but can be any value subject to your operating system naming requirements.) For example, if you preprocess an ESQL source file named `esqltest.cc`, the preprocessor generates a C source file named `esqltest.c`.

Although you can view your preprocessed source file to understand how it works, do not modify it.

- Each time you preprocess your ESQL application during the development cycle, a new preprocessed source file is created. This can overwrite your modifications.
- If you incorrectly modify ESQL function calls or variable assignments, you could corrupt your database.
- Calling sequences that the preprocessor generates might change with future releases.

For more information on using the SQL preprocessor, see Chapter 3, "Building and Running ESQL Applications."

1.6.3 Compiling the Preprocessed Source Files

After generating your C source files, compile them with your C compiler. At this point, you can specify compile-time options, some of which can affect your ESQL application. For example, you can set an application option that defines the maximum number of host language variables you can reference in a single ESQL statement.

For more information on compiling ESQL source files, see Chapter 3, "Building and Running ESQL Applications."

1.6.4 Linking the ESQL Executable

After successfully compiling your source files, you have a corresponding number of object files. Link your object files with Progress object files to produce an ESQL executable. The Progress objects you link with depend on the type of executable you are building:

- Large client with the SQL Compiler
- Small client

Except for Windows small clients, you can link your executable using a link script that the PROBUILD utility generates. For Windows small clients, you use your own link script. For more information, see Chapter 3, "Building and Running ESQL Applications." For information on the PROBUILD utility, see the *Progress Client Deployment Guide*.

Functional Layers of an ESQL Executable

Figure 1–7 shows the functional layers that constitute a large client and small client executable.

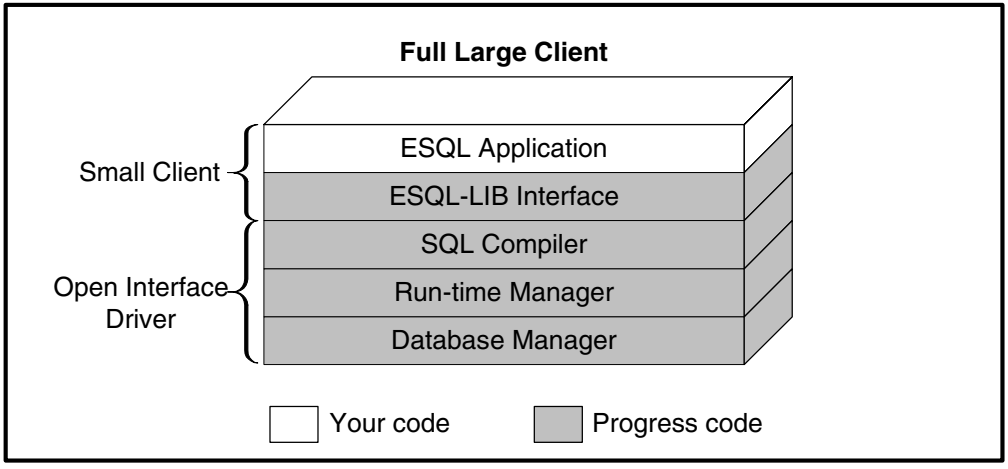

Figure 1–7: Large Client with Compiler-functional Layers

Figure 1–7 demonstrates an ESQL executable that is built from your ESQL application code and Progress code (provided in object form). The Progress code consists of the following functional layers:

- **ESQL-LIB interface** — Progress/ESQL support library. This is part of any small or large client executable.
- **SQL Compiler** — Code that compiles your SQL statements, in some cases, generating r-code files for those SQL statements. If a statement is compiled into r-code, it is not recompiled unless you delete its r-code file. This is part of any large client executable and the OID.

- **Run-time manager** — Code that runs compiled SQL statements (r-code files). This is part of any large client executable and the OID.
- **Database manager** — Code that handles all database access and much of the processing involved in query resolution. This is part of any large client executable and the OID.

NOTE: Communications layers (not shown) are also available for all ESQL executables and the OID.

For more information on linking ESQL source files, see Chapter 3, "Building and Running ESQL Applications."

2

Programming with ESQL

How you program with ESQL depends on whether your application requires only static ESQL, dynamic ESQL, or a combination of both. If you specify your SQL statements at compile time and use only static ESQL, you must write your application using Progress/ESQL, and then preprocess your source files to generate the required ESQL-LIB function calls.

To specify SQL statements at run time (requiring dynamic ESQL), you can write a combination of Progress/ESQL and direct ESQL-LIB function calls that you preprocess to generate the remaining calls to ESQL-LIB, or you can completely implement the application with direct ESQL-LIB function calls without preprocessing.

To complete an application using either static or dynamic ESQL, you write some direct ESQL-LIB function calls to manage the interaction between your application and Progress. This chapter provides information on performing all of these tasks, including:

- An overview of ESQL programming
- Elements of Progress/ESQL
- Starting and managing an ESQL session
- Using dynamic ESQL
- Using dynamic parameter markers
- Dynamic ESQL statements for prepared execution

2.1 Programming Capabilities and Options

The objective of ESQL is to allow a C Language application to interact with Progress using SQL. To this end, ESQL enables you to access Progress/SQL plus additional capabilities only in ESQL. These capabilities include requirements and options unique to its C programming environment.

Using ESQL, you can execute any data manipulation language (DML) or data definition language (DDL) statement available in Progress/SQL. In addition, you can execute some DDL statements not available in Progress/SQL, for example, the CREATE SCHEMA statement. For more information on Progress/SQL DML and DDL statements, see the *Progress SQL-89 Guide and Reference*.

2.1.1 ESQL Sessions

You execute all ESQL statements within an ESQL session, except for declarations for ESQL data and condition handling. To begin an ESQL session, log into Progress. To end the session, log out using the appropriate ESQL-LIB functions.

You can only run one ESQL session in a single executable. Once you end an ESQL session, you cannot start another one without restarting your executable. However, you can connect and disconnect Progress databases any number of times within a single ESQL session.

2.1.2 Static and Dynamic ESQL

You can write your application using static ESQL, dynamic ESQL, or a combination of both. Static ESQL provides a basic syntax for specifying Progress/SQL statements directly in your C Language code. You can use static ESQL when you know the exact contents of your SQL statements at compile time.

Dynamic ESQL includes additional statements and C data structures for feeding run-time SQL statements to Progress and returning results to your application. Use dynamic ESQL when your application cannot determine the contents of its SQL statements until run time.

2.1.3 Progress/ESQL and ESQL-LIB

Progress/ESQL provides a portable and consistent syntax for specifying SQL statements in your application. The SQL preprocessor converts Progress/ESQL into calls to ESQL-LIB functions to implement your static or dynamic ESQL statements. However, there are additional calls to ESQL-LIB you must make yourself, even when using Progress/ESQL. These calls provide additional functionality not specified in the SQL standard, but that are necessary or useful for running an ESQL session. Examples of tasks that you can accomplish only through direct calls to ESQL-LIB are logging in and out of the session and retrieving ESQL status messages.

2.2 Writing an ESQL Application

Once you determine your application objectives and design how it interacts with your SQL database(s), you are ready to begin coding your ESQL application. These are typical steps to implement an ESQL session using Progress/ESQL:

1 ♦ Declare the C variables that you need for your application, including those needed to execute ESQL statements and return error conditions.

2 ♦ Declare the actions that the program takes when ESQL statements execute with exceptional conditions.

3 ♦ Log into the ESQL-LIB interface to begin your ESQL session using the `sqllogin()` function.

4 ♦ Execute additional ESQL-LIB functions and Progress/ESQL statements to connect databases, invoke SQL statements, and disconnect databases.

5 ♦ Log out of the ESQL-LIB interface to end your ESQL session using the `sqllogout()` function.

Figure 2–1 illustrates these steps with a sample ESQL application written to query the sports database using static ESQL.

NOTE: For simplicity, this example assumes that all database connection parameters are specified on the command line and acted on by `sqllogin()` as part of Step 3. However, a more flexible approach is to control database access from within the application by selectively connecting and disconnecting databases using the ESQL-LIB functions, `sqlconn()` and `sqldiscon()`. For more information, see the section "Starting and Ending an ESQL Session."

```
 main(argc,argv)
        int argc;
        char **argv;
  {
/*1*/  /* Declare variables for error returns. */
       long SQLCODE;
       char *msg;

       /* Declare variables for ESQL statements. */
       EXEC SQL BEGIN DECLARE SECTION;
       char nam[30];
       long custnum;
       EXEC SQL END DECLARE SECTION;

/*2*/  /* Declare error goto point. */
        EXEC SQL WHENEVER SQLERROR GOTO err;

/*3*/  /* Log in to Progress to start an ESQL session. */
       if (sqllogin( argc-1, &argv[1], "", "") == -1)
       {
           printf("Failed to login to Progress ESQL!\n");
           while ((msg = (char *) sqlgetmsg()) != (char *) 0)
               printf("%s\n",msg);
           exit(2);
       }

/*4*/  /* Query and display all customer names and numbers. */
       EXEC SQL DECLARE x CURSOR FOR
           SELECT name, cust\-num FROM customer;
       EXEC SQL OPEN x;

       EXEC SQL WHENEVER NOT FOUND GOTO done1;
       while (1)
       {
           EXEC SQL FETCH x INTO :nam, :custnum;
           printf("%-22s%-7ld\n",nam,custnum);
       }

/*5*/  /* Log out of Progress to end the ESQL session. */
   err:
       printf("Error %-3ld occured.\n", SQLCODE);
       while ((msg = (char *) sqlgetmsg()) != (char *) 0)
           printf("%s\n",msg);

   done1:
       sqllogout(1);
   }
```

Figure 2–1: Sample ESQL Application (Not Preprocessed)

1. ESQL requires the SQLCODE or SQLSTATE variable declaration to return error codes to your application.

2. The application specifies a branching label to handle error conditions. In general, you declare any label for error branching in each function that contains ESQL statements.

3. As part of the login process, ESQL connects to the database using Progress startup parameters from the command line passed in `argv`.

4. The backslash (\) in cust\-num is required to specify hyphenated identifiers, because SQL does not permit the hyphenated identifiers that are allowed in Progress. However, you can preprocess and execute your application with the ESQL No Padding (-esqlnopad) startup parameter to avoid this restriction. The application also defines a standard exit point (done1) to terminate loops that operate on rows. The variable names, nam and custnum , are preceded by a colon (:).

5. The application logs out of Progress for both error returns (at err) and normal returns (at done1).

Figure 2–2 shows the same application module after it is preprocessed.

(1 of 2)

```
#include "proesql.h"
   static SQLRQHDL *sql0 = (SQLRQHDL *) 0;
   main(argc,argv)
        int argc;
        char **argv;
   {
/*1*/  /* Declare variables for error returns. */
       long SQLCODE;
       char *msg;

       /* Declare variables for ESQL statements. */
       /* EXEC SQL BEGIN DECLARE SECTION; */
       char nam[30];
       long custnum;
       /* EXEC SQL END DECLARE SECTION; */

/*2*/  /* Declare error goto point. */
       /* EXEC SQL WHENEVER SQLERROR GOTO err; */

/*3*/  /* Log in to Progress to start an ESQL session. */
       if (sqllogin( argc-1, &argv[1], "", "") == -1)
       {
           printf("Failed to login to Progress ESQL!\n");
           while ((msg = (char *) sqlgetmsg()) != (char *) 0)
               printf("%s\n",msg);
           exit(2);
       }
/* Query and display all customer names and numbers. */
      /* EXEC SQL DECLARE x CURSOR FOR
          SELECT name, cust\-num FROM CUSTOMER; */
      /* EXEC SQL OPEN x; */
      {
/*sqltxt: DECLARE x CURSOR FOR SELECT name, cust-num FROM CUSTOMER*/
  static char sqltxt[] = {
  'D','E','C','L','A','R','E',' ','x',' ','C','U','R','S','O','R',
' ','F','O','R',' ','S','E','L','E','C','T',' ','n','a','m','e',',',
' ','c','u','s','t','-','n','u','m',
' ','F','R','O','M',',','C','U','S','T','O','M','E','R',0};
static char sqlfmt[] = {0};
          sqlcdbind(&SQLCODE, (char *)0);
          sqlrequest(&sql0,(int)0x04,sqltxt,"x",sqlfmt,(char**)0);
          if (SQLCODE < 0) goto err;
      }
```

(2 of 2)

```
 /* EXEC SQL WHENEVER NOT FOUND GOTO done1; */
      while (1)
      {
          /* EXEC SQL FETCH x INTO :nam, :custnum; */
          {
  /*sqltxt: FETCH x INTO esqlo0, esqlo1
    sqlfmt: c30l,*/
  static char sqltxt[] = {
  'F','E','T','C','H',' ','x',' ','I','N','T','O',' ',
'e','s','q','l','o','0',',',' ','e','s','q','l','o','1',0};
static char sqlfmt[] = {'c','3','0','l',',',0};
              static char *sqlv[3];
              sqlv[0] = (char *) nam;
              sqlv[1] = (char *) &custnum;
              sqlv[2] = (char *)0;
              sqlcdbind(&SQLCODE, (char *)0);
              sqlfetch(sql0,(int) 0x08,sqltxt,sqlfmt,sqlv);
              if (SQLCODE == 100) goto done1;
              if (SQLCODE < 0) goto err;
          }
          printf("%-22s%-7ld\n", nam, custnum);
      }
/*5*/  /* Log out of Progress to end the ESQL session. */
   err:
       printf("Error %-3ld occured.\n", SQLCODE);
       while ((msg = (char *) sqlgetmsg()) != (char *) 0)
           printf("%s\n",msg);

   done1:
       sqllogout(1);
   }
```

Figure 2–2: Sample ESQL Application (Preprocessed)

A practical ESQL application typically includes much more SQL than this, but these fundamental steps form the backbone of any ESQL application. The remaining sections in this chapter explain the techniques available to expand and modify this basic approach.

2.3 Elements of Progress/ESQL

This section introduces Progress/ESQL and the requirements for using it in your application.

2.3.1 Basic Progress/ESQL Syntax

An ESQL statement has the following general syntax:

SYNTAX

```
EXEC SQL statement-spec ;
```

Each ESQL statement begins with the EXEC SQL prefix to identify it as an ESQL statement to the SQL preprocessor. The *statement-spec* component can be any Progress/SQL statement (with the necessary modifications) or other ESQL statement elements. When *statement-spec* is a Progress/SQL statement, you are using static ESQL.

The maximum size of an ESQL statement is 4K.

2.3.2 Reserved Words

The Compiler reserves many keywords in Progress/SQL and the Progress 4GL. You cannot use these reserved words as table or column names when you create or modify an SQL schema. For information on the reserved Progress keywords, see the keyword index of the *Progress Language Reference*.

2.3.3 Application Function Names

When writing application functions, choose function (or entry point) names that do not conflict with Progress and ESQL-LIB entry point names. If you are planning to run in the large client model, it is even more important because the entire Progress global name space is exposed to your application.

In most cases, if you have an entry point in your code with the same name as a Progress entry point, the linker identifies duplicate entry point symbols with an error message. However, sometimes the linker replaces one entry point with the other, and does not display a message.

To avoid global naming problems, name your functions using the following syntax:

SYNTAX

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

function-name

Any valid C function name that meets the following requirements:

- It must not begin with “sql”. The preprocessor uses “sql” as a prefix for all symbols that it generates.
- It must have a maximum length at least four characters less than the maximum name length that your compiler allows.
- It must be compatible with your development environment.

No Progress entry point name ends with “_USR”, so this suffix guarantees that your application has no conflicts with Progress global symbols. For example, you can include a function called `calc_USR()` without any conflict.

NOTE: Building your application using the small client model reduces, but does not eliminate, the likelihood of name conflicts described in this section.

2.3.4 SQL Operators

Table 2–1 lists the operators you can use in ESQL. These are the ANSI standard SQL operators, except as noted. These are also a subset of the operators available to the Progress 4GL. For more information on these operators, see the *Progress SQL-89 Guide and Reference*.

Table 2–1: Progress/SQL Operators *(1 of 2)*

SQL Operator	Operation
- (unary negative)	Reverses the sign of a numeric expression.
+ (unary positive)	Preserves the sign of a numeric expression.
/ (division)	Divides one numeric expression by another to produce a decimal numeric result.
* (multiplication)	Multiplies two numeric expressions.

Table 2–1: Progress/SQL Operators *(2 of 2)*

SQL Operator	Operation
- (DATE subtraction)[1]	Subtracts a number of days from a date to produce a DATE result, or subtracts one date from another to return the number of days between the two dates.
- (subtraction)	Subtracts one numeric expression from another.
+ (DATE addition)[1]	Adds a number of days to a date to produce a DATE result.
+ (concatenation)	Concatenates two character strings or character expressions.[1]
+ (addition)	Adds two numeric expressions.
<	Returns a true value if the first of two expressions is less than the second.[2]
< =	Returns a true value if the first of two expressions is less than or equal to the second.[2]
>	Returns a true value if the first of two expressions is greater than the second.[2]
> =	Returns a true value if the first of two expressions is greater than or equal to the second.[2]
=	Returns a true value if two expressions are equal.[2]
< >	Returns a true value if two expressions are not equal.[2]
NOT	Reverses the true or false value of an expression.
AND	Returns a true value if each of two logical expressions is true.
OR	Returns a true value if either of two logical expressions is true.

[1] Non-standard SQL.

[2] Returns an unknown value if either operand is NULL.

NOTE: The SQL standard specifies that a NULL value in the result set of a subquery sets the truth value of the subquery relation to FALSE if the relation specifies that all values must satisfy the relational operator (if the relational operator is qualified by the ALL option). The subquery relation can yield an unknown (?) value if no other value satisfies the relational operator. For more information on the results of subquery relations, see the *Progress SQL-89 Guide and Reference*.

2.3.5 Progress Function Support

Table 2–2 lists the Progress functions that are supported in ESQL. Any other Progress functions permitted in Progress/SQL are not permitted in ESQL. However, ANSI standard SQL aggregate functions (such as AVG and COUNT) are permitted in Progress/ESQL.

When you reference Progress identifiers (for example, table fields (or columns) or function names) that contain hyphens in Progress/ESQL, precede the hyphen with a backslash (\) to escape the hyphen. If you do not insert a backslash before the hyphen, the SQL preprocessor interprets the hyphen (-) as a minus sign because hyphens are not permitted in ANSI standard SQL identifiers.

Table 2–2: Progress Function Support in ESQL *(1 of 2)*

_SERIAL-NUM	GET-CODEPAGES	PROMSGS
ABSOLUTE	GET-COLLATIONS	PROPATH
ALIAS	IF-THEN-ELSE	PROVERSION
ASC	INDEX	R-INDEX
CAN-DO	INTEGER	RANDOM
CAN-FIND	KEYWORD	RECID
CAPS	KEYWORD-ALL	REPLACE
CHR	LC	RIGHT-TRIM
CONNECTED	LDBNAME	ROUND
CONTAINS	LEFT-TRIM	ROWID
CURRENT-LANGUAGE	LENGTH	SDBNAME
CURRENT-VALUE	LIBRARY	SQRT

Table 2–2: Progress Function Support in ESQL *(2 of 2)*

DATE	LOG	STRING
DAY	LOOKUP	SUBSTITUTE
DBCODEPAGE	MACHINE-CLASS	SUBSTRING
DBCOLLATION	MATCHES	TIME
DBNAME	MEMBER	TODAY
DBRESTRICTIONS	MONTH	TRIM
DBTYPE	NEXT-VALUE	TRUNCATE
DBVERSION	NUM-ALIASES	WEEKDAY
DECIMAL	NUM-DBS	YEAR
ENCODE	NUM-ENTRIES	–
ENTRY	OPSYS	–
ETIME	OS-DRIVES	–
EXP	OS-GETENV	–
FILL	PDBNAME	–

2.3.6 3GL Access to Database Schema Triggers

Database schema triggers are 4GL procedures that Progress invokes automatically when it performs database operations such as row updates, creates, and deletes. They are stored in the database independently from application logic and respond equally to all applications that access the database. For more information on schema triggers, see the *Progress Programming Handbook*.

If you access a Progress database via ESQL, database schema triggers execute appropriately. For example, the DELETE statement executes the DELETE trigger and the INSERT statement executes the CREATE trigger.

However, ESQL applications that execute database schema triggers cannot compile the trigger procedure 4GL code; you must compile all schema trigger procedures prior to running your ESQL application. Also, ESQL applications cannot execute schema triggers that contain certain invalid 4GL keywords.

Invalid 4GL Keywords

An ESQL application cannot execute a schema trigger that executes 4GL user interface and operating system I/O operations. If ESQL tries to execute such a schema trigger, it generates an error message at the SQL statement that caused the trigger to execute and sets the ESQL error variables to the default error codes: SQLCODE to -1; SQLSTATE to 50000. (For more information on SQLCODE and SQLSTATE, see the section "Error Handling.") The error message identifies the trigger name and the line number within the trigger procedure that contains the invalid keyword.

Table 2–3 lists the 4GL keywords that are invalid in schema triggers executed by ESQL applications.

Table 2–3: Invalid 4GL Keywords in ESQL Schema Triggers

ALIAS	ENABLE	READKEY
APPLY	HIDE	SCROLL
BELL	IMPORT	SEARCH
CHOOSE	INPUT	SEEK
CLEAR	KEY-VALUE	SET
CLOSE	LOAD	STATUS
COMPILE	NEXT-PROMPT	SYSTEM-DIALOG
CONNECT	ON	UNDERLINE
CREATE *widget*	OS escapes	UNLOAD
DDE	OUTPUT	UP
DELETE *widget*	PAGE	UPDATE
DISABLE	PAUSE	VIEW
DISCONNECT	PROMPT-FOR	WAIT-FOR
DISPLAY	PUT	–
DOWN	PUT-VALUE	–

Large Client Applications and Host Language Call Functions

For ESQL large client applications, schema triggers can use the Host Language Call (HLC) interface. When you use the CALL statement to execute 3GL functions in a schema trigger executed by an ESQL large client application, the following HLC functions are disabled:

- `proscopn()`
- `proscclr()`
- `prosccls()`

However, you can use the HLC `promsgd()` function to add messages to the list of messages returned by the ESQL `sqlgetmsg()` function. For more information on HLC and HLC functions, see the *Progress External Program Interfaces* manual.

2.3.7 ESQL Data Types

When you access table columns or supply values for dynamic parameter markers in Progress/ESQL, the data types of SQL data objects must be assignment compatible with the host language variables in your ESQL application. See Appendix B, "ESQL Data Type Compatibilities." This section lists the SQL data types that Progress supports.

SYNTAX

```
CHARACTER [ ( n ) ]
```

A character string of *n* characters. You can abbreviate CHARACTER to CHAR. In Progress/SQL, the length is used only to form the default display format ("x(n)"), since character values in Progress are variable-length.

NOTE: In dynamic ESQL, the length value is returned to the *LENGTH* structure member describing a CHARACTER column in the SQL descriptor area. For more information, see the "Using Dynamic Embedded SQL" section.

SYNTAX

```
INTEGER
```

`INTEGER`

A whole number from -2,147,483,648 to 2,147,483,647, inclusive. You can abbreviate INTEGER to INT.

SYNTAX

```
SMALLINT
```

SMALLINT

Maps to the INTEGER data type.

SYNTAX

```
DECIMAL [ ( m [ , n ] ) ]
```

DECIMAL

A decimal number up to 50 digits in length. The precision (*m*) specifies the number of significant digits. The scale (*n*) determines the number of digits to the right of the decimal point. The scale can be up to 10 digits (15 when running with the ANSI SQL (-Q) startup parameter). Progress/SQL truncates the value to the specified scale before storing it in the column. You can abbreviate DECIMAL to DEC.

SYNTAX

```
FLOAT [ ( m ) ]
```

FLOAT

Maps to the DECIMAL data type. By default, the FLOAT data type receives a sliding point format. The value of (*m*) is currently ignored.

SYNTAX

```
DOUBLE PRECISION
```

DOUBLE PRECISION

Maps to the DECIMAL data type. The DOUBLE PRECISION data type receives a sliding point format.

SYNTAX

```
DATE
```

DATE

A date from 1/1/32768 B.C. through 12/31/32767 A.D. You can specify dates in this century with the format *mm/dd/yy* or with *mm/dd/yyyy*. Dates in other centuries require the format *mm/dd/***[-]***yyyyy* or *dd/mm/***[-]***yyyyy*, depending on your data format.

SYNTAX

```
LOGICAL
```

LOGICAL

YES/NO or true/false values.

SYNTAX

```
REAL
```

REAL

Maps to the DECIMAL data type.

SYNTAX

```
NUMERIC [ ( m [ , n ] ) ]
```

NUMERIC

Equivalent to DECIMAL.

NOTE: RAW data types are not supported in Progress/ESQL. An attempt to retrieve data of this type will result in an error.

When you create or alter tables in your SQL schema, specify these SQL data types for the columns of your tables. Progress, in turn, defines these columns in terms of the five core Progress data types: CHARACTER, DATE, DECIMAL, INTEGER, and LOGICAL.

When you access SQL tables for input or output, your application must provide host language variables or data structure members whose data types correspond to the columns and dynamic parameters referenced in your ESQL statements. For information on the basic C data types available for application storage, see the "Host Language Variables" section in this chapter.

Progress provides a variety of data conversions between different core Progress, SQL, and C data types, depending on whether you are using static or dynamic ESQL. For more information, see Appendix B, "ESQL Data Type Compatibilities."

2.3.8 WHENEVER Statement

The WHENEVER statement tells the preprocessor to generate code that checks the value of SQLCODE or SQLSTATE for a specified condition after each SQL request. The WHENEVER statement has the following syntax:

SYNTAX

```
EXEC SQL WHENEVER { SQLERROR | NOT FOUND } [ { GOTO | GO TO } [ : ] label
; CONTINUE ]
```

When using SQLCODE, specify the SQLERROR condition so the preprocessor generates code that checks for a negative value in SQLCODE. Specify the NOT FOUND condition so the preprocessor generates code that checks for a value of +100 in SQLCODE (no row satisfies the SQL request). For more information, see the "Error Handling" section.

You can specify an action for the SQLERROR or NOT FOUND condition. The target of a GOTO action (*label*) is a standard C Language label name. The program branches to this label when it detects the specified condition. For the SQLERROR and NOT FOUND conditions, the default action is CONTINUE, which causes the program to ignore the condition and continue with the next statement.

The specified condition and action remains in effect until you specify another WHENEVER statement for the same condition. Any new action overrides the previous one. However, you can explicitly test the value of SQLCODE or SQLSTATE in a C statement at any point in your program without affecting the current WHENEVER statement specifications.

In this example, the WHENEVER statement signals the preprocessor to generate code that transfers program control to err when an SQL error condition occurs.

```
EXEC SQL WHENEVER SQLERROR GOTO err;
```

In the following example, the program continues execution whenever no record is found to satisfy an SQL request, such as a searched SELECT or cursor FETCH.

```
EXEC SQL WHENEVER NOT FOUND CONTINUE;
```

The parameter values of `sqlcdbind()` depend on the variables declared in an SQL declare section. Table 2–4 lists the parameter values of sqlcdbind for the three possible conditions.

Table 2–4: Parameter Values of sqlcdbind()

Syntax	Condition
`sqlcdbind(&SQLCODE, SQLSTATE);`	When both SQLCODE and SQLSTATE are declared in an SQL declare section.
`sqlcdbind((long *)0, SQLSTATE);`	When SQLSTATE is declared in an SQL declare section, but SQLCODE is not. See below for WHENEVER statement behavior under this condition.
`sqlcdbind(&SQLCODE, (char *)0);`	When SQLSTATE is not declared in an SQL declare section. SQLCODE may or may not be declared in an SQL declare section for this condition.

If SQLSTATE is declared in an SQL declare section but SQLCODE is not, the preprocessor generates the following from the WHENEVER SQL ERROR statement:

```
if (SQLSTATE[0] != '0' || SQLSTATE[1] > '2') ...
```

In either of the other conditions listed in Table 2–4, the preprocessor generates the following:

```
if (SQLCODE < 0)...
```

If SQLSTATE is declared in an SQL declare section but SQLCODE is not, the preprocessor generates the following from the WHENEVER NOT FOUND statement:

```
if (SQLSTATE[0] == '0' && SQLSTATE[1] == '2') ...
```

In either of the other conditions listed in Table 2–4, the preprocessor generates the following:

```
if (SQLCODE == 100)...
```

2.3.9 Progress/ESQL DDL Statements

Progress/ESQL provides access to all DDL statements available in Progress/SQL. In addition, Progress/ESQL provides the CREATE SCHEMA statement which creates an entire schema in a single statement. The CREATE SCHEMA statement has the following syntax:

SYNTAX

```
EXEC SQL CREATE SCHEMA AUTHORIZATION db-user {CREATE-TABLE-statement |
CREATE-VIEW-statement } ... ; GRANT-statement
```

The options allow you to create tables AND views, and grant table and column privileges in a connected database with the logical name that *db-user* specifies. The options can refer to tables and views that earlier options of the same CREATE SCHEMA statement create. For more information on the options, see the *Progress SQL-89 Guide and Reference*.

This statement uses the logical database name (*db-user*) as a user ID and password to allow your application to create multiple schemas, possibly with the same table and view names, associated with different users. To distinguish the schemas in Progress, each user corresponds to a different logical database.

For example, suppose that your application creates a customer table in a schema with the user name *jon* and attempts to access the table with the following SELECT statement fragment:

```
... SELECT * FROM customer ...
```

To resolve any ambiguity between the customer table in the jon schema and a table with the same name in other schemas, ESQL interprets the fragment as follows.

```
... SELECT * FROM jon.customer ...
```

2.4 Host Language Variables

With ESQL, you can pass data between your application and ESQL using host language variables. Host language variables are specially declared C Language variables that you can specify in Progress/ESQL statements. You can use host language variables to exchange data between your application and SQL tables with both static and dynamic ESQL statements. You can also use them to provide values for dynamic parameter markers in dynamic ESQL. (For more information on dynamic parameter markers, see the "Using Dynamic Parameter Markers" section in this chapter.)

2.4.1 Host Language Variables and ESQL-LIB

Host language variables have meaning only to Progress/ESQL and the preprocessor. If your application only makes direct calls to dynamic ESQL-LIB functions, use the SQLDA structure (SQL Descriptor Area), a special data structure that defines your application data.

2.4.2 Host Language Variables and Dynamic ESQL

Dynamic ESQL uses SQL Descriptor Areas to exchange data between your application and ESQL. If you use SQL Descriptor Areas, you do not have to declare host variables in your application even if you use the preprocessor. The preprocessor converts any host language variables that you declare and use in dynamic ESQL into SQL descriptor references. For more information on using SQL descriptor areas and host language variables in dynamic ESQL, see the "Using Dynamic Embedded SQL" section.

The SQL Declare section has the following syntax:

SYNTAX

```
EXEC SQL BEGIN DECLARE SECTION;

    host-variable-declaration;
    host-variable-declaration;
              .
              .
              .
    host-variable-declaration;

EXEC SQL END DECLARE SECTION;
```

Each *host-variable-declaration* in the section is a standard C Language variable declaration, as in the following example:

```
EXEC SQL BEGIN DECLARE SECTION;
    char name[30];
    long  custnum, maxcred, hicustnum;
    char city[20], state[3];
EXEC SQL END DECLARE SECTION;
```

Like any C variable declaration, you can place the SQL Declare section anywhere before the ESQL statements that refer to it and in the same function or module scope as these statements. For an example, see Step 1 of Figure 2–1 and Figure 2–2.

2.4.3 ESQL Standard Type Definitions and Conversions

ESQL provides a set of predefined type definitions you can use to define host language variables for date, decimal, and logical table columns. These definitions are in the `proesql.h` header file that comes with your ESQL installation, and provide these data types:

- sqldate

  Defines a 13-character variable to hold a date string including the null terminator. The date moved into an sqldate variable for input must be a string in the format "*mm/dd/*[-]*yyyy*" or "*dd/mm/*[-]*yyyy*", depending on the setting of the Date Format (-d) startup parameter at run time.

- sqldec

  Defines a 53-character variable to hold a decimal string value: 50 characters for the digits, one for the sign, one for a decimal point, and one for the null terminator.

- sqlbool

  Defines a one-character variable to hold a binary value of 0 (Progress FALSE) or 1 (Progress TRUE).

In addition to these standard ESQL data types, you can use a number of the standard C data types. The actual data types you can use for a host variable depends on the SQL data type of the corresponding column or parameter marker, and whether you are using static or dynamic ESQL.

ESQL performs a variety of data conversions depending on the ESQL style you are using. For more information on ESQL data type correspondences and conversions, see Appendix B, "ESQL Data Type Compatibilities."

NOTE: Using the C int data type for host language variables in either static or dynamic ESQL is nonstandard, and the preprocessor flags any instances of it with a warning. Use either short or long.

2.4.4 Using Host Language Variables

You use host language variables as storage to retrieve values from and insert values into SQL table columns, and to provide values for parameter markers in dynamic ESQL statements. Follow these rules when you use host language variables in ESQL:

- You must declare host language variables in an SQL Declare section before you reference them in other ESQL statements. ESQL uses this section to identify the variables and their data types for data conversion.
- In ESQL statement references, you prepend each host language variable name with a colon (:) and separate each host language variable specification from the next with a comma (,), as in this ESQL statement. For an example in context, see Step 4 of Figure 2–1 and Figure 2–2:

```
EXEC SQL FETCH x INTO :nam, :custnum;
```

- Each host language variable specification can include an indicator variable to store NULL values. For more information, see the "Indicator Variables" section.
- You cannot specify host variables with C Language type definitions other than those defined in the ESQL header file (`proesql.h`) provided with your ESQL installation.
- You must specify data types for host variables that are compatible with the SQL data types of their corresponding columns or parameter markers. For more information on the compatibilities between Progress/SQL and C data types, see Appendix B, "ESQL Data Type Compatibilities."
- You can use only scalar variables or character arrays (strings) as host language variables.
- You cannot use qualified or array-subscripted variables (that is, structure members or array elements) as host language variables.

- You cannot use C Language preprocessor constants (#define... constants) to declare or reference host variables.
- The Progress/ESQL C preprocessor does not support the following features of the new ANSI C standard:
  - Trigraphs cannot occur anywhere in the source file.
  - String continuations, indicated by quoted strings separated by white space, may not be used for string literals within the SQL-specific sections of a program. They may be used in the Progress 4GL code or in the non-SQL specific sections of the C code.

2.5 Indicator Variables

Progress/ESQL supports NULL values in columns and expressions. Progress represents NULL (unknown) values with a question mark (?). However, because there is no data representation for NULL or unknown values in C, they are represented with indicator variables. Using indicator variables, you can exchange NULL values between host language variables of any data type and ESQL in a consistent manner.

NOTE: Progress/SQL provides the NULL keyword to represent NULL values in interactive SQL, and also provides indicator variables for the same purpose. Progress/SQL statements do not recognize the unknown value (?) as a literal expression.

Indicator variables also inform the host program that character string variables have been truncated (the database value is too long to fit in the host language variable). When a character string variable is truncated and an indicator variable is present, the actual length of the character column in the database is stored in the indicator variable.

An indicator variable provides information about a host language variable that is not representable in the host language variable value itself. In ESQL, indicator variables can represent:

- Null values in host language variables or table columns.
- The full length of the data in a character column after the data is returned and truncated in a host language variable.

2.5.1 Declaring and Specifying Indicator Variables

You declare indicator variables like host language variables in an SQL Declare section, and declare them as C numeric types. Indicator types can be long, short, sqldec, float, or double. For information on declaring host language variables, see the "Host Language Variables" section in this chapter.

Use this syntax to specify indicator variables in an ESQL statement wherever you can specify host language variables:

SYNTAX

```
:host-language-variable-ref [ [ INDICATOR ] :indicator-variable-ref ]
```

A host language variable expression can optionally include an indicator variable. Do not separate corresponding host language and indicator variables with a comma. However, you do use a comma to separate one host language variable expression (including any indicator variable) from the next.

2.5.2 Using Indicator Variables to Exchange NULL Values

Indicator variables in ESQL work much like indicator variables in Progress/SQL. When the indicator variable is set to -1, the SQL value of the host language variable specification is NULL. When the indicator variable is set to 0, the SQL value of the host language variable specification is in the host language variable itself.

To assign (or input) a NULL value to the column or parameter marker corresponding to a host language variable, set the indicator variable to -1 before executing the statement that references it. Similarly, when you retrieve a NULL column value into a host language variable, ESQL sets the corresponding indicator variable to -1. If you provide an indicator variable, check the indicator variable value to determine what value has actually been returned. For NULL SQL values, whether input or output, the value of the host language variable is meaningless.

If you try to retrieve a NULL value to a host language variable without a corresponding indicator variable, ESQL returns an error with SQLCODE set to -43, or SQLSTATE set to 22 002.

To assign any input value other than NULL, set the host language variable to the value you want to assign and set any corresponding indicator variable to 0. To check for any output value other than NULL, first check to see that the indicator value is 0 before checking the host language variable for a non-NULL value.

2.5.3 Using Indicator Variables to Retrieve NULL Values

This code fragment retrieves output to the host language variable :lowmark and the corresponding indicator variable mc.

```
EXEC SQL SELECT credit\-limit INTO :lowmark INDICATOR :mc
           FROM customer
          WHERE cust\-num = :lowcustnum;
```

If the selected credit-limit contains a NULL value, mc is set to -1.

Similarly, cn is set to -1 in the following code fragment for each instance of the cust-num column that contains a NULL value:

```
EXEC SQL DECLARE x CURSOR FOR SELECT cust\-num, name, city, state
                              FROM customer;
EXEC SQL OPEN X;
EXEC SQL FETCH x INTO :custnum INDICATOR :cn, :nam, :city, :state;
```

2.5.4 Using Indicator Variables to Set NULL Values

In this example, the indicator variable, costind, is set to -1 before the SQL statement executes. When the UPDATE statement executes, it sets the price column in any rows that satisfy the search condition to the NULL value:

```
costind = -1;

EXEC SQL UPDATE item SET price = :costvar INDICATOR :costind
         WHERE item\-num > 1000;
```

2.6 Error Handling

Error status information is useful to applications since it often lets them convert failure to success.There are four mechanisms to retrieve status information from Progress/ESQL:

- SQLCODE
- SQLSTATE
- Progress error messages
- Function return codes

Depending on the requirements of your application's error handling, you may decide to use some or all of these mechanisms.

2.6.1 SQLCODE Variable

The SQLCODE variable is an integer whose value indicates the success or failure of the most recently executed SQL statement.

Table 2–5 shows the SQLCODE return code values and their meanings.

Table 2–5: SQLCODE Return Code Values

Return Code	Meaning
0	The statement executed successfully and no exceptional conditions occurred.
+100[1]	No rows were found to satisfy the request (see Table 2–6).
< 0	An error occurred. For example, an update operation violated a specified constraint, such as NOT NULL or UNIQUE. See Table 2–7 for the individual error codes and meanings.

[1] Other positive values are reserved for future informational and error messages.

Table 2–6 shows the meaning of a +100 return code for each SQL statement where this return code can occur.

Table 2–6: Meanings of the +100 Return Code

Statement	Meaning
FETCH	There is no next row.
SELECT INSERT . . . SELECT	There are no rows to select.
Searched UPDATE[1]	No rows were found to update.
Searched DELETE[1]	No rows were found to delete.

[1] Positioned UPDATE and DELETE return an error if no row is present to satisfy the SQL request.

Table 2–7 lists and describes the specific error code values for SQLCODE.

Table 2–7: SQLCODE Error Code Values *(1 of 2)*

Error Code	Meaning
-1	General compile or run-time error. Usually results from incorrect syntax.
-2	Cardinality violation. A SELECT...INTO... statement returned more than one row, or a subquery preceded by an arithmetic operator (such as =) returned more than one row.
-11	The schema for the database has changed since the r-code for this request was generated. You must recompile the request. If the SQL Compiler is available, it recompiles automatically; otherwise, this error returns.
-16	The UPDATE...WHERE CURRENT OF... statement was not entered in the correct form.
-18	A cursor request was attempted, but the cursor has not been opened.
-21	A **CTRL-C** rolled back the transaction.
-26	An ESQL-LIB routine was called from within the **CTRL-C** interrupt handler.
-28	Error extracting value from host language variable.
-29	Error updating a host language variable.
-30	The database system caused the current transaction to be rolled back.
-40	Error extracting a host language variable.
-41	Error updating a host language variable.
-42	An attempt was made to convert between incompatible types. (For example, an attempt to convert from an SQLDADATE input to a Progress INTEGER data type is never valid.)
-43	An attempt was made to store a NULL value in a host language variable without setting an associated indicator variable.
-44	A converted value does not fit in the destination variable or field.
-45	An attempt was made to convert a string value to another type, but the current value of the string is not valid for that type.

Table 2–7: SQLCODE Error Code Values *(2 of 2)*

Error Code	Meaning
-46	HLI_NOTCAP. ESQL is not capable of performing the requested operation.
-99	A fatal error occurred from which Progress cannot recover; you can make no further calls to ESQL-LIB except to the `sqlgetmsg()` function.

Using SQLCODE in an Application

If you are using the SQL preprocessor, you declare a variable named SQLCODE as a long C within the scope of any function that contains any Progress/ESQL statements except the WHENEVER, BEGIN DECLARE SECTION, or END DECLARE SECTION statements:

```
long SQLCODE;
```

You receive source code compilation errors if you do not declare the SQLCODE variable as a long C data type. You do not have to declare it within the SQL DECLARE SECTION of your application. However, it is recommended that you do so.

The SQL preprocessor uses the address of this declared variable as an argument to an ESQL-LIB function, `sqlcdbind()`. By placing this function call prior to many of the calls to the ESQL-LIB interface in the generated output, the SQL preprocessor notifies the interface where to deposit the status information.

If you are not using the SQL preprocessor, or if you are using it and are accessing some of the ESQL-LIB functions that have no counterpart in Embedded SQL (especially the dynamic ESQL-LIB functions), you may still use SQLCODE as the mechanism to retrieve status information about the ESQL-LIB function calls being made. In this case, you declare a long C integer in your application and pass the address of that variable to the `sqlcdbind()` function. You do not have to declare the variable with the name SQLCODE.

2.6.2 SQLSTATE Variable

The SQLSTATE variable is a five-character string whose value indicates the outcome of the most recently executed SQL statement. SQLSTATE can contain only digits and uppercase letters. The first two characters of SQLSTATE indicate a class; the following three characters indicate a subclass. Class codes are unique, but subclass codes are not; the meaning of a subclass code depends on the class code in which it is contained.

Class codes starting with 0-4 or A-H indicate error conditions and exceptions that are defined in the latest ANSI SQL Standard (ANSI X3.135-1992). Subclass codes for conditions specified in this same standard also start with 0-4 and A-H.

Class codes starting with 5-9 or I-Z indicate Progress-specific error conditions and exceptions. There are no subclasses for these class codes.

Table 2–8 shows the SQLSTATE return code values that the ESQL entry points return to the application under the listed conditions.

Table 2–8: SQLSTATE Error Code Values *(1 of 3)*

Error Code	Condition
00 000	The request completed with unconditional success.
02 000	Returned when a FETCH returns no row because the cursor was already at the end of the set, when a SELECT produces an empty table, or the condition specified in an INSERT, searched DELETE, or searched UPDATE is not satisfied.
07 001	Using clause does not match dynamic parameters. Occurs when the SQLDA supplied to `sqldynexec()` is bad because of one of the following: The COLCNT is less than the number of dynamic parameter markers but greater than zero, a TYPE field contains an unknown value, or a DATA field contains a NULL pointer.
07002	Using clause does not match target specification. Occurs when the SQLDA supplied to `sqldynftch()` is bad because of one of the following: The CONLCNT is less than the number of result columns but greater than zero, a TYPE field contains an unknown value, or a DATA field contains a NULL pointer.
07 004	Using clause required for dynamic parameters. The prepared request contained dynamic parameter markers, but either the SQLDA pointer supplied to `sqldynexec()` was NULL, or the COLCNT was zero.
07 006	Restricted data type attribute violation. An attempt was made to store a value to a data type that is incompatible with the data type of the source (for example, an attempt to store an SQLDATE to an SQL integer).
07 008	Invalid descriptor count. The COLCNT field of an SQLDATE supplied to `sqldynexec()` or `sqldynftch()` contained a value greater than the SQLN field.

Table 2–8: SQLSTATE Error Code Values *(2 of 3)*

Error Code	Condition
07 009	Invalid descriptor index. The parameter supplied to `sqlald()` was invalid because it was either less than zero or greater than the system-defined maximum number of SQLDATE indexes.
08 001	Client unable to establish connection to database; `sqlconn()` failed.
08 003	Connection does not exist; `sqldisconn()` failed because the database was not currently connected.
21 000	Cardinality violation. A SELECT... INTO or a subquery returned more than one result row.
22 002	Null value, no indicator parameter. The application tried to fetch a null value but did not supply an indicator variable in which to indicate that the value is null.
22 003	A numeric value is too large to be stored in a destination data type without losing significant digits. Currently, this state may not be returned if the value results from calculating an expression.
22 005	An error occurred while trying to convert a value to the data type of the destination (for example, trying to store the string “ABCD” to an integer).
22 012	Division by zero.
22 024	An unterminated string value was supplied to ESQL from the application. Either a host language variable declared as character[*n*] contained *n* non-null characters, or an SQLDA was supplied with a LENGTH field of *n* and there were *n*+1 non-null characters to which the DATA field pointed.
23 000	Integrity constraint violation. A statement attempted to change a table in such a way that the resulting table would violate the constraints on the table. Either a column declared as NOT NULL was set to NULL, or a column declared as UNIQUE was set to a duplicate value.
24 000	An attempt was made to do something to a cursor that is invalid given the current state of the cursor. All of the following attempts result in this state: OPEN an already OPEN cursor, CLOSE an unopened cursor, FETCH from an unopened cursor, DELETE from an unopened or unpositioned cursor, or UPDATE an unopened or unpositioned cursor.

Table 2–8: SQLSTATE Error Code Values *(3 of 3)*

Error Code	Condition
26 000	Invalid SQL statement identifier. This is returned from `sqldynexec()` or `sqldyndesc()` if it is called with a stale request handle. There is no prepared request associated with the request handle; the statement identifier is invalid.
33 000	Invalid SQL descriptor name. Returned from `sqldynftch()`, `sqldyndesc()`, and `sqldyndescinp()` if the SQLDA pointer supplied to the entry point is null or invalid.
37 000	Syntax error or access violation in PREPAREd request. See state 42 000. Same as state 42 000 except this state is returned for statements that were PREPAREd, while 42000 is returned for all other statements.
40 000	Indicates that the current transaction has been rolled back. This may be returned by any function that accesses a database.
42 000	Indicates that there was either a syntax error in the request, or the request could not be carried out for one of these reasons: the table, view, or index does not exist or the user does not have sufficient privilege to carry out the operation. The state is used for several different error conditions for security reasons. The user should not be able to differentiate between a request that failed because there was a syntax error or a request that failed because a table does not exist.
50 000	Indicates that a statement failed but does not indicate the reason for the failure. It is a default error state, used when none of the more specific error states are appropriate.
UN SET	Unset state. At the beginning of every entry point, the ESQL sets SQLSTATE to UNSET. Before ESQL returns control to the application, it sets the SQLSTATE to the correct value. The application should never be returned to this state. If so, it indicates that ESQL failed to properly set SQLSTATE. However, the value of SQLCODE is still reliable.
ZZ 000	A fatal, unrecoverable error occurred. No more calls to the ESQL entry points can be made. Currently returned by `sqllogout()` only.

Using SQLSTATE in an Application

SQLSTATE is the preferred status reporting mechanism, although the SQL preprocessor does not require it. If you use SQLSTATE with the SQL preprocessor, declare SQLSTATE as follows within the SQL DECLARE SECTION in the scope of any function that contains any Progress/ESQL statements except the WHENEVER, BEGIN DECLARE SECTION, or END DECLARE SECTION statements:

```
char SQLSTATE[6];
```

The SQL preprocessor uses the address of this declared variable as an argument to an ESQL-LIB function, `sqlcdbind()`. By placing this function call prior to many of the calls to the ESQL-LIB interface in the generated output, the SQL preprocessor notifies the interface where to deposit the status information. If the SQLSTATE variable declaration is not found in the SQL DECLARE SECTION, then the SQL preprocessor does not pass any SQLSTATE value to the `sqlcdbind()` function.

If you are not using the SQL preprocessor, or if you are using it and are accessing some of the ESQL-LIB functions that have no counterpart in Embedded SQL (especially the dynamic ESQL-LIB functions), you may still use SQLSTATE to retrieve status information about the ESQL-LIB function calls being made. In this case, declare the 6-byte character array in your application and pass the address of that variable to the `sqlcdbind()` function. You do not have to declare the variable with the name SQLSTATE.

2.6.3 Progress Error Messages

In conjunction with SQLCODE and SQLSTATE, Progress/ESQL generates Progress error messages under error or warning conditions. The text of these messages is available with the ESQL-LIB `sqlgetmsg()` function. Multiple messages may be generated on the execution of any SQL statement. Generally, all ESQL-LIB functions can potentially generate error messages.

2.6.4 Function Return Codes

Most normalized functions return status values. The exceptions include functions that either return other functional data or do not return any values at all. See Chapter 4, “ESQL-LIB Reference” for more information on general diagnostics.

2.7 Starting and Ending an ESQL Session

Before you can invoke most ESQL-LIB functions and Progress/ESQL statements, call the `sqllogin()` function to log into the ESQL-LIB interface and start your ESQL session. The functions and statements you can invoke before starting an ESQL session include:

- The `sqlgetmsg()` function to display any current messages in the Progress message queue. For more information, see the "Error Handling" section.
- The WHENEVER statement to define standard responses to SQL conditions. For more information, see the "WHENEVER Statement" section.
- The BEGIN DECLARE SECTION and END DECLARE SECTION statements to declare host language variables. For more information, see the "Host Language Variables" section in this chapter.

2.7.1 Starting a Session

The `sqllogin()` function initiates communication with ESQL-LIB and optionally passes any startup parameters you (or the application user) specify to Progress, including any database connection parameters. It can also specify an optional application name used to generate filenames for SQL r-code files compiled during execution.

You can call `sqllogin()` only once. The `sqllogin()` function returns 0 if it successfully logs in to ESQL-LIB and connects any specified databases. If it fails for any reason (including failure to connect databases), `sqllogin()` returns -1 and ends your ESQL session. The only ESQL-LIB function you can call after `sqllogin()` fails is `sqlgetmsg()` . Once `sqllogin()` fails, you can make no other calls to ESQL-LIB (including `sqllogin()`) without exiting to the operating system and restarting your application.

NOTE: Because `sqllogin()` ends your ESQL session if it fails, your application cannot recover if `sqllogin()` fails to connect a specified database. To maintain your ESQL session after such a failure, connect to databases using the `sqlconn()` function, and pass to it only client startup parameters that have no effect on database connection and run-time behavior. For more information on `sqlconn()` and database connection, see the "Connecting and Disconnecting Databases" section. For more information on client startup parameters, see the *Progress Startup Command and Parameter Reference*.

Specifying Progress Startup Parameters

The startup parameters that you can specify to `sqllogin()` for the small client model are different than for the large client model. Following are the parameters you can specify for the small client model:

- The Open Interface Driver (-SV) startup parameter and its list of connection parameters. For more information, see Chapter 3, "Building and Running ESQL Applications."
- The common ESQL startup parameters. For more information, see Chapter 3, "Building and Running ESQL Applications."
- Database connection parameters. For more information, see the "Connecting and Disconnecting Databases" section.

Following are the parameters you can specify for the large client model:

- The common ESQL startup parameters. For more information, see Chapter 3, "Building and Running ESQL Applications."
- Database connection parameters. For more information, see the "Connecting and Disconnecting Databases" section.
- Progress session-wide client startup parameters, which are any Progress client startup parameters that are not database connection parameters. For more information, see the *Progress Startup Command and Parameter Reference*.

  NOTE: For the small client model, these parameters are specified when starting the Open Interface Broker or Driver. For more information, see Chapter 3, "Building and Running ESQL Applications."

You can specify Progress startup parameters to `sqllogin()` using three techniques:

- Pass the parameters directly in a character string.

  For example, if you log in connecting to a database named `wildlife` with the user name **Jon** and the password **noj**, your call to `sqllogin()` might look like this:

```
int ret;
ret = sqllogin(0,(char **)0, -db wildlife -U Jon -P noj", "");
```

- Pass the parameters in a parameter file that you specify in a character string.

  Use the -pf startup parameter to pass the name of a parameter file to Progress. If the name of your file is `my_options.pf`, your call to `sqllogin()` might look like this:

```
int ret;
ret = sqllogin(0,(char **)0, -pf my_options.pf", "");
```

- Pass the parameters from command line arguments.

  In a C program, the argv argument specified for the `main()` function points to an array of pointers to the command line arguments that are passed to your application. The argc argument is the number of command line arguments. With most C compilers, the first element (element 0) points to the name of the executable image. Omit the first element to only pass arguments that are meaningful to Progress. If you use this technique, your call to `sqllogin()` might look like this:

```
main(argc,argv)
int argc;
char *argv[ ];
{
  int ret;
  ret = sqllogin(argc-1,&argv[1],"","");
```

  NOTE: If you use argv to specify your startup parameters from the command line, strip any application-defined parameters before passing the argument list to `sqllogin()`.

You can also use these techniques together, as shown in the following example:

```
main(argc,argv)
int argc;
char *argv[ ];
{
  int ret;
  ret = sqllogin(argc-1,&argv[1],"-db wildlife -U Jon -P noj","");
```

Here, you pass `sqllogin()` both startup parameters from the command line and startup parameters in a character string. ESQL-LIB passes all these startup parameters to Progress with the command line parameters last. Any parameter settings that the application user specifies at startup replace any equivalent settings in the command string. Thus, you can specify default parameter settings that the user can override.

Specifying an Application Name

When you log in with `sqllogin()`, you can specify an application name. ESQL uses the first two characters of this name plus a random hexadecimal number to generate the filename for each SQL r-code file compiled during execution.

To associate run-time compiled r-code files with the sections of your application in which they are compiled, specify a different application name at the start of each section using the `sqlapplname()` ESQL-LIB function.

If you use a null string as an application name (shown as the fourth `sqllogin()` argument in the previous code fragments), ESQL generates the entire filename for each r-code file that it compiles using a random hexadecimal number string.

2.7.2 Ending a Session

To log out of the ESQL-LIB interface and end your ESQL session, call the `sqllogout()` function. This function closes all open connections to databases and logs out of the ESQL-LIB interface, ending your ESQL session (however, it does not exit your application). Once you call `sqllogout()`, you can make no other calls to ESQL-LIB except `sqlgetmsg()` without exiting to the operating system and restarting your application.

NOTE: The `sqllogout()` function also rolls back any uncommitted work. For more information on managing transactions, see the "Managing Transactions in ESQL" section.

2.8 Connecting and Disconnecting Databases

You can connect to databases in your ESQL application in two ways:

- Any time during an ESQL session, by calling the ESQL-LIB `sqlconn()` function.
- At login time, by calling the ESQL-LIB `sqllogin()` function.

Of the two techniques, calling `sqlconn()` provides greater flexibility because you are already logged in to an ESQL session when you make the call. If an error occurs while calling `sqlconn()`, you can recover and retry the database connection without ending your ESQL session. Also, using both the `sqlconn()` and `sqldiscon()` functions, you can connect and disconnect databases within the same session as often as you want. However, if a database connection attempt fails while calling `sqllogin()`, your ESQL session effectively ends and you must restart your application to retry the connection. For more information on `sqllogin()`, see the "Starting and Ending an ESQL Session" section.

2.8.1 Connecting Databases

Following is the syntax for `sqlconn()`:

SYNTAX

```
int
sqlconn ( pparms )
  char  *pparms;    /* INPUT */
```

The character string you pass to `sqlconn()` must specify the physical filenames of databases you want to access in your program. You can also specify any Progress database connection parameters in the string you specify in the Progress 4GL CONNECT statement. You can also specify the Parameter File (-pf) startup parameter to indicate the name of a parameter file that contains your list of database connection parameters.

The `sqlconn()` function only accepts database connection parameters and ignores Progress parameters (such as the Date Format (-d) startup parameter) that do not affect connection and run-time behavior for a specific database.

When you connect to a database, it receives a default logical name. This is the physical name of the database file without the `.db` extension. You can also assign a logical name using the Logical Database Name (-ld) startup parameter. You can then reference this logical name in your ESQL statements and `sqldiscon()` function as an application standard. For more information on database logical names, see the *Progress Programming Handbook*.

See the *Progress Startup Command and Parameter Reference* for a list of parameters you can use when connecting databases by calling `sqllogin()` or `sqlconn()`.

Accessing Multiple Databases in ESQL

You can access multiple Progress and non-Progress databases in an ESQL application. To access multiple databases, you only need to specify the appropriate startup parameters when you log in and when you connect to the databases using `sqlconn()`.

By default, Progress can connect up to five databases at a time. If you want to connect to more than five databases, specify the number of databases using the Number of Databases (-h) startup parameter. The maximum number of databases to which you can connect is 240. For a large client, you can pass -h to `sqllogin()` when you start your ESQL session. For a small client, specify -h as a startup parameter when you start the Open Interface Broker (OIB) or manual Open Interface Driver (OID). For more information on the Number of Databases (-h) startup parameter, see the *Progress Startup Command and Parameter Reference*. For more information on the OIB and OID, see Chapter 3, "Building and Running ESQL Applications."

To actually connect your databases, you can make a separate call to `sqlconn()` for each database or a single call to connect them all at once. If you call `sqlconn()` once to connect multiple databases, include the -db option before each database name except the first. For example, the following `sqlconn()` call connects to databases customer and order:

```
sqlconn("customer -db order");
```

Using Database Name Qualifiers

When connecting to multiple databases in an ESQL program, you may need to qualify table names to avoid ambiguous references. Progress/SQL uses a two-part naming convention. The syntax to qualify a table name for a database is *database-name.table-name*. However, to qualify the columns in selection lists for ambiguous table references, specify a correlation name for the table to provide a single name to qualify each ambiguous column name. You can specify a correlation name following each qualified or unqualified table name in a table reference. You separate the table name and correlation name in the table reference with the AS keyword:

SYNTAX

```
SELECT ... FROM [database-name.]table-name AS correlation name
  [ , [database-name.]table-name AS correlation name ] ...;
```

You can use two-part references of the form *correlation-name.column-name* to uniquely identify ambiguous column references in selection lists, as shown in this example:

```
EXEC SQL DECLARE x CURSOR FOR
    SELECT firstcust.name, firstcust.cust\-num,
           secondcust.name, secondcust.cust\-num
    FROM first.customer AS firstcust, second.customer AS secondcust;
```

2.8.2 Disconnecting Databases

When you are finished accessing a database, use the `sqldiscon()` function to disconnect it.

Following is the syntax to call `sqldiscon()`:

SYNTAX

```
int
sqldiscon  ( pldbname )
  char  *pldbname;    /* INPUT */
```

The following sqldiscon() call disconnects the currently connected database with the logical name “customer”:

```
sqldiscon("customer");
```

NOTE: Work done while a database is connected is not committed when you call sqldiscon(). You commit work using the COMMIT WORK statement before you disconnect the database, or before exiting from your program. Otherwise, the disconnection does not occur and your open transaction is rolled back. Databases do not disconnect until a transaction ends. Therefore, any disconnections pend until the application encounters a COMMIT WORK or ROLLBACK WORK statement or an sqllogout() call (which automatically rolls back all work). If the transaction does not complete and you reconnect the database, the disconnect never actually occurs. For more information on transaction management, see the “Managing Transactions in ESQL” section in this chapter.

2.9 Managing Transactions in ESQL

The way Progress handles transaction processing differs between ESQL and Progress/SQL. Progress/SQL uses the same block-oriented transaction management model as the Progress 4GL. However, in Embedded SQL, a transaction begins when you call sqllogin() and continues until you execute one of these statements:

- COMMIT WORK
- ROLLBACK WORK
- Any DDL statement

You can include COMMIT WORK and ROLLBACK WORK statements anywhere in your application. Use the COMMIT WORK and ROLLBACK WORK statements frequently since all updated rows are locked until one of these statements is encountered. See the esqldemo.cc and dyndemo.cc sample applications provided with your installation for information on how the COMMIT WORK and ROLLBACK WORK statements are used in ESQL.

NOTE: If you call sqllogout() before committing the current transaction, the transaction is rolled back.

2.9.1 Committing Work

The COMMIT WORK statement commits changes you make to the database. When you execute the COMMIT WORK statement, all open cursors are closed, the resources associated with them are released, and changes you made since the last COMMIT WORK or ROLLBACK WORK statement become permanent. Use the COMMIT WORK statement to commit work before exiting from your program, since the `sqllogout()` call rolls back work by default.

The COMMIT WORK statement has the following syntax:

SYNTAX

```
EXEC SQL COMMIT WORK ;
```

2.9.2 Rolling Back Work

The ROLLBACK WORK statement cancels changes made to the database since the last COMMIT WORK or ROLLBACK WORK statement. When you execute the ROLLBACK WORK statement, all updates made by the transaction are cancelled and all open cursors are closed.

The ROLLBACK WORK statement has the following syntax:

SYNTAX

```
EXEC SQL ROLLBACK WORK ;
```

2.9.3 Using DDL Statements in Transactions

When you include a DDL statement in an ESQL application, the current transaction is committed when the statement successfully executes. Therefore, do not execute a DDL statement during a transaction in which you want the option of executing a ROLLBACK WORK statement, because the DDL statement automatically causes the transaction to commit.

2.10 Checking ESQL Run-time Privileges

Progress automatically checks privileges in ESQL programs at compile time, in the same way it does for standard Progress and Progress/SQL procedures. However, Progress does not check privileges at run time. To prevent unauthorized users from running your Embedded SQL programs, use the Progress CAN-DO function at appropriate points in your application.

The CAN-DO function returns a 1 (true) or a 0 (false) based on the value of a character field (a privilege field) that contains a comma-separated list of user ID string values. If the specified user IDs (or the user ID of the application) has permission from the list, it returns 1; otherwise it returns 0. To retrieve and store the CAN-DO function value, define a variable of type sqlbool as the target of a singleton SELECT or FETCH.

For more information on how user ID values specify permissions, see the CAN-DO function entry in the *Progress Language Reference*. For information on using tables to store activities-based permissions, see the security sections of the *Progress Programming Handbook*.

2.10.1 Using the CAN-DO Function in a Singleton SELECT

To use the CAN-DO function in a singleton SELECT, you can invoke it with the following syntax:

SYNTAX

```
EXEC SQL SELECT CAN\-DO( privilege-field [ , user-list ] )
  INTO :target-variable
  FROM privilege-table
  WHERE record-select-clause ;
```

privilege-field

The name of the privilege field associated with a permission you want to check (such as the _Can-Read field in the Progress _File schema table). It can exist either in a Progress schema table or in an activities-based permissions table that you create.

user-list

A comma-separated list of user IDs. If you do not specify this value, the CAN-DO function uses the user ID of the ESQL session, which you can also specify with the ESQL keyword USER.

target-variable

A host language variable of type sqlbool to receive the value of the CAN-DO function.

privilege-table

The name of a table that contains privilege fields.

record-select-clause

A selection clause that specifies a specific record of the *privilege-table* that you are querying.

This example selects a record from the _File schema table when the value of the _File-name schema field equals “customer”, tests the value of the _Can-Write schema field, and sets the host language variable `cando` to the result:

```
EXEC SQL BEGIN DECLARE SECTION;
  sqlbool cando;
EXEC SQL END DECLARE SECTION;
                    .
                    .
                    .
EXEC SQL SELECT CAN\-DO(_Can\-Write, USER) INTO :cando
           FROM _File
          WHERE _File\-name = "customer";
```

You subsequently test the value of the host language variable to determine what action to take next. For example, if the CAN-DO function returns a 0, you can display an error message from your program:

```
if (cando == 0)
{
  writeln("Sorry, you are not authorized to run this program.");
  return -1;
}
                        .
                        .
                        .
```

2.10.2 Using the CAN-DO Function in a WHERE Clause

You can also use the CAN-DO function in the WHERE clause of an SQL statement. For example, you might want to define security constraints for selected rows in a table.

In this example, user privileges for accessing a table with payroll information are stored in the access column of each row. In this case, the access column might grant access privileges to the employee (and manager) whose payroll information the row contains:

```
EXEC SQL DECLARE x CURSOR FOR
    SELECT * FROM payroll WHERE CAN-DO(payroll.access, USER);
                          .
                          .
                          .
```

2.11 Managing Multi-user Applications

To coordinate concurrent access to records in multi-user applications, ESQL statements use a record locking scheme based on the Progress record locking phrases. You do not insert Progress record locking phrases directly into ESQL statements. Record locking phrases are derived from the corresponding SQL syntax elements.

Table 2–9 shows the Progress record locks and explains their effects.

Table 2–9: Progress Record Locking Phrases

Lock Type	Effect
NO-LOCK	Rows are not locked. A row is read even if another application holds an EXCLUSIVE-LOCK on it. You cannot update rows read with NO-LOCK.
SHARE-LOCK	Rows are read in anticipation of a possible update. A row is not read if another application holds an EXCLUSIVE-LOCK on it.
EXCLUSIVE-LOCK	Rows are locked until the end of the transaction. Another application cannot read rows using SHARE-LOCK or EXCLUSIVE-LOCK.

Table 2–10 lists the types of SQL statements that obtain locks and the locking options associated with them.

Table 2–10: SQL Statements and Associated Record Locks

Embedded SQL Statement	Associated Record Lock
Non-cursor SELECT statements (SELECT INTO)	NO-LOCK only. You can see any uncommitted changes to the database.
OPEN cursor statements[1]	SHARE-LOCK by default. Records retrieved through a cursor use SHARE-LOCK while being read and are converted to EXCLUSIVE-LOCK if updated or deleted. To read records with the NO-LOCK option, specify FOR READ ONLY in the associated DECLARE CURSOR statement or use the -NL startup parameter to set the default locking mode to NO-LOCK.[2] A cursor declared FOR UPDATE will SHARE-LOCK the records retrieved. The records are updated to EXCLUSIVE-LOCK when they are fetched.
Searched UPDATE statements (UPDATE WHERE *condition*) Positioned UPDATE statements (UPDATE WHERE CURRENT) Searched DELETE statements (DELETE WHERE *condition*) Positioned DELETE statements (DELETE WHERE CURRENT)	EXCLUSIVE-LOCK only.

[1] Cursor SELECT statements do not obtain locks until the cursor is opened.

[2] In dynamic ESQL, specify the lock criteria in the SQL request string following the cursor specification. For more information, see the "Executing Dynamic Cursor Specifications" section in this chapter.

2.11.1 Application Lock Management

By default, ESQL waits indefinitely on behalf of your application for a locked database resource. However, ESQL allows your application to specify a different response when it encounters a locked resource by calling the `sqlsetlogoption()` function. The `sqlsetlogoption()` function allows you to specify your own callback function that tells ESQL what to do when it encounters a locked resource.

If you supply a callback function, ESQL calls the callback function each time it encounters a resource lock and proceeds according to the value that you return from the function. Depending on the value you return, ESQL can:

- Wait for the lock to clear indefinitely (the default behavior).
- Continue waiting for the lock to clear, but make another call to the callback function.
- Stop waiting for the lock and return to your application with SQLCODE set to -1 (error).

For more information on how to define and use your lock management callback function, see the `sqlsetlogoption()` reference entry in Chapter 4, "ESQL-LIB Reference."

NOTE: By default, ESQL waits for a locked resource rather than returning an error. So if a lock is detected and then immediately released, the application receives the record. Otherwise, if an error is always returned in record lock situations, then even when the resource becomes immediately available, the application cannot obtain it.

Handling Interrupts in ESQL Large Clients

In the large client model, if you press **CTRL-G** during execution, the current SQL statement is aborted (if you are accessing the database) or the next SQL statement is aborted (if you are not accessing the database). You can cancel a resource lock wait during large client operation using the **CTRL-G** interrupt. (For information on cancelling resource lock waits during small client operation, see the "Managing Multi-user Applications" section.) In any case, ESQL detects the interrupt, rolls back any open transaction, and closes any open cursors.

NOTE: The lock callback function mechanism described in the previous section is the preferred method for handling resource lock issues.

When ESQL detects the **CTRL-C** interrupt, it executes a function named `sqlctlc()`. The default `sqlctlc()` function is a dummy entry point that takes no action. However, you can create your own `sqlctlc()` function to print an error message, set a global flag that you check in your program, or perform any other action that your application defines-except calling other ESQL-LIB functions. If you attempt to call other ESQL-LIB functions from within `sqlctlc()`, ESQL returns an error.

CAUTION: Do not define your own **CTRL-C** interrupt handler in place of `sqlctlc()`. If you do specify another handler, ESQL does not roll back the current transaction and close open cursors when the interrupt occurs. However, you can add any additional interrupt handling functionality to `sqlctlc()` without preventing ESQL from handling transactions and cursors appropriately.

2.12 Using Dynamic Embedded SQL

Dynamic ESQL is a variant of Embedded SQL that allows your ESQL application to create SQL statements at run time. Unlike static ESQL, you do not have to define the text of dynamic ESQL statements at application development time. Instead you can enter them directly, generate them from user prompts, or generate them at run time using a third-party application such as a spreadsheet.

Dynamic ESQL also allows you to prepare, compile, and execute (run) statements that contain parameter markers. Parameters markers are place holders in SQL expressions for values that your application provides when it executes a statement. Using parameter markers, you can prepare a statement once and execute the same SQL statement repeatedly, substituting different values for the parameters each time, without incurring repeated compiler overhead.

2.12.1 Differences Between Dynamic and Static ESQL

The run-time capabilities of dynamic ESQL require additional syntax and differ in functionality from static ESQL. The differences include:

- A different technique to specify SQL statements, using a run-time SQL request string.
- A different set of techniques to execute SQL statements, using additional Progress/ESQL statements and ESQL-LIB functions.
- Changes and restrictions on how you form some SQL statements.

Specifying SQL Statements

In static ESQL, you specify all SQL statements as part of the Progress/ESQL statement syntax, which the preprocessor converts to r-code references or hard-coded character strings for compilation at run-time. In dynamic ESQL, you specify SQL statements in SQL *request strings* that you can build from scratch at run time.

A *request string* is a C character string that contains the text of an SQL statement, but does not include an EXEC SQL prefix, a semicolon (*;*) statement terminator, or an embedded host variable name. (The prefix and terminator apply only to Progress/ESQL statements.) For more information, see the "Elements of Progress/ESQL" section.

The work to specify, compile, and execute SQL statements in dynamic ESQL is divided between the user, your application, and ESQL. To manage this division of labor, dynamic ESQL allows the user (or application) to specify most elements of an SQL request in the request string, while restricting other elements to the application. The application controls all elements that handle the exchange of data between the application and SQL statements, the management of cursor specifications that the user or application might construct, and the compilation and execution of SQL statements themselves.

Thus, dynamic ESQL request strings cannot contain the following:

- DECLARE CURSOR, OPEN, FETCH, or CLOSE statements used to manage cursor specifications.

  Instead, ESQL provides dynamic forms of these statements. Your application can use these and other Progress/ESQL statements to perform all data and cursor specification management, as well as to compile and execute the SQL statements.

- References to embedded host variable names.

- SQL comments.

- Dynamic parameters in a context that prevents the data type of the surrounding expression from being inferred.

NOTE: Dynamic ESQL provides no way for the user or your application to directly specify a singleton SELECT (SELECT...INTO statement) that returns a single row of data. Instead, your application must execute SELECT statements as cursor specifications. These can also include SELECT statements that return a single row of data for retrieval using the dynamic FETCH statement. For more information, see the "Executing Dynamic Cursor Specifications" section in this chapter.

To support run-time SQL statements, dynamic ESQL provides additional Progress/ESQL statements and ESQL-LIB functions. These functions support techniques that let you specify, compile, and execute SQL statements at run time. Much of this functionality allows the exchange of data between your application and the SQL statements. In static ESQL, this functionality is not needed because your application knows the content of all SQL statements at development time.

2.13 Executing Dynamic ESQL

Dynamic ESQL provides two techniques for forming and executing SQL statements:

- Immediate execution
- Prepared execution

2.13.1 Immediate Execution

You can use this technique for any SQL request that you can complete in one statement and that does not return data to your application. Immediate execution compiles and executes the SQL statement in the request string each time you invoke it. A request string can contain the following SQL statements for immediate execution:

- DDL statements
- Searched UPDATE statements
- Searched DELETE statements
- INSERT statements
- Positioned UPDATE statements (if the corresponding open dynamic cursor can be found)
- Positioned DELETE statements (if the corresponding open dynamic cursor can be found)

For UPDATE and INSERT statements, specify all column data values in the request string.

To implement immediate execution, ESQL provides the Progress/ESQL EXECUTE IMMEDIATE statement and the supporting `sqldynrq()` ESQL-LIB function. For more information, see the "Using Immediate Execution" section.

2.13.2 Prepared Execution

You can use this technique for any SQL request that you can (or must) compile before executing. Prepared execution compiles each SQL request and executes the compiled request any number of times without recompiling. This technique is necessary for dynamic cursor specifications and statements that contain dynamic parameter markers. Thus, a request string can contain the following SQL statements for prepared execution:

- DDL statements (possible, but with no benefit over immediate execution)
- Searched UPDATE statements

- Searched DELETE statements
- INSERT statements
- Cursor specifications (cursor SELECT statements with or without dynamic parameter markers)
- Positioned UPDATE statements
- Positioned DELETE statements

For UPDATE and INSERT statements, dynamic parameter markers provide an alternative means to set column values from your application. For cursor specifications, the dynamic FETCH statement returns column data to your application.

To implement prepared execution, ESQL provides a number of Progress/ESQL statements and supporting ESQL-LIB functions. For more information, see the "Using Prepared Execution" section in this chapter.

2.13.3 Elements of Dynamic ESQL Execution

The following sections describe how to use immediate execution and the more complex techniques for prepared execution, including dynamic parameter markers.

For a basic dynamic ESQL application example, see the `dyndemo.cc` Progress/ESQL source file provided in your ESQL installation directory. For a complete dynamic ESQL application example that includes dynamic parameter support, see `dprmdemo.cc`. This example uses a combination of ESQL and ESQL-LIB. For a complete example that calls ESQL-LIB directly and also includes parameter marker support and data type conversion examples, see the `clidemo.c` C source file in the same directory.

2.14 Using Immediate Execution

You typically use immediate execution for any SQL statements that do not return data to your application and do not need to execute more than once per ESQL session. DDL statements are common candidates for immediate execution. But you can use this technique for any of the statements that are appropriate for immediate execution listed in the "Executing Dynamic ESQL" section in this chapter.

NOTE: SQL statements for immediate execution cannot contain dynamic parameter markers.

ESQL provides the EXECUTE IMMEDIATE statement to perform immediate execution using the following syntax:

SYNTAX

```
EXEC SQL EXECUTE IMMEDIATE request-string;
```

request-string

A C character string expression specifying the SQL statement to execute. This character string must contain direct SQL. It cannot contain the EXEC SQL prefix, the semicolon (;) terminator, or embedded host variable names used in ESQL statements. See the "Differences Between Dynamic and Static ESQL" section for more information and additional restrictions on the request-string.

ESQL compiles the SQL statement in the *request-string* at run time, but does not save the resulting r-code. If the same statement is encountered again, it is recompiled.

The following code fragment shows an example using the EXECUTE IMMEDIATE statement:

```
long SQLCODE;
char *prequest;
            .
            .
            .
prequest = "DROP TABLE mytable";
EXEC SQL EXECUTE IMMEDIATE prequest;
```

In this example, the character pointer request is set to the string "DROP TABLE mytable", a DDL statement. Then the EXECUTE IMMEDIATE statement executes the statement. The preprocessor converts this statement to a call to the `sqldynrq()` ESQL-LIB function:

```
static SQLRQHDL *sql0 = (SQLRQHDL *) 0;
            .
            .
            .
long SQLCODE;
char *prequest;
            .
            .
            .
prequest = "DROP TABLE mytable";
/* EXEC SQL EXECUTE IMMEDIATE prequest; */
{
    sqlcdbind(&SQLCODE, (char *)0);
    sqldynrq(&sql0,prequest);
}
```

This function compiles and executes the SQL request. For information on the `sqldynrq()` function, see Chapter 4, "ESQL-LIB Reference."

NOTE: For statements that you can execute using immediate execution, you can also use prepared execution. See the "Using Prepared Execution" section in this chapter.

2.15 Using Prepared Execution

Use prepared execution for the following types of SQL statements:

- Cursor specifications, with which you can retrieve one or more rows of a table based on a set of search criteria.

  NOTE: Multiple cursors are supported for static and dynamic ESQL.

- Any SQL statements that contain dynamic parameter markers.

For a complete list of statements with which you can use prepared execution, see the "Executing Dynamic ESQL" section in this chapter. You can use prepared execution for any SQL statement except singleton SELECTs. However, there is no benefit in doing so for statements that you can invoke using immediate execution and only need to execute once in an ESQL session. This is especially true for DDL statements because of how prepared execution operates.

2.15.1 Elements of Prepared Execution

Prepared execution of an SQL request includes two basic phases:

1. A preparation phase, in which the statement is compiled using the PREPARE statement (or the `sqldynprep()` ESQL-LIB function).

2. An execution phase, in which the statement is executed using either the dynamic OPEN or EXECUTE statement (both supported by the `sqldynexec()` ESQL-LIB function).

NOTE: Immediate execution combines these two phases to execute an SQL request into one step. Without any further steps to manage execution, the request is prepared, executed, and discarded. To execute the same request again, repeat both phases. This is efficient for statements (such as DDL statements) that perform one task without having to exchange data with the application. This is also why prepared execution provides no benefit for such statements, where the extra data and execution management is not required or used.

Depending on the type of SQL request you are executing, there can be additional tasks to set up input and output data structures, and to manage execution of the request. ESQL provides a number of Progress/ESQL statements and ESQL-LIB functions to carry out these tasks. For a description of the Progress/ESQL statements provided for prepared execution, see the "Dynamic ESQL Statements for Prepared Execution" section in this chapter. For more information on ESQL-LIB functions, see Chapter 4, "ESQL-LIB Reference."

2.15.2 Data Exchange in Prepared Execution

Prepared execution allows you to exchange data between your application and ESQL using SQL descriptor areas or host language variables. An *SQL descriptor area* (SQLDA) is a data structure that describes and provides storage for input or output column data, and also for parameter marker values. After preparing an SQL request, there are several ESQL statements and functions that you can use to allocate storage for the request and initialize any SQLDAs to complete execution of the request.

If you use host language variables for a request, ESQL ultimately uses an SQLDA to complete the data exchange. If you specify host language variables for dynamic column input or output or for parameter marker values in certain dynamic ESQL statements, the preprocessor generates a default SQLDA that references the specified host language variables for data storage. For more information, see the "SQL Descriptor Area" section in this chapter.

2.15.3 Forms of Prepared Execution

The forms of prepared execution are related to the type of SQL request you are executing. There are two general forms in ESQL:

- Dynamic cursor specification
- All other dynamic SQL statements

In addition, each of these forms can include dynamic parameter markers, depending on the SQL statement being prepared.

2.15.4 Prepared Execution for Dynamic Cursor Specification

Follow these general steps to execute a dynamic cursor specification:

1. Prepare the cursor SELECT using the PREPARE statement (or `sqldynprep()`), establishing an SQL request handle for the statement.
2. Allocate an SQLDA to retrieve rows from the result set. It must be large enough for the number of columns in the cursor specification.

   There are several techniques available to complete this step. You can skip this step if you have the preprocessor allocate the SQLDA for you from a host language variable list specified later in this procedure.
3. If you complete Step 2, initialize the column attributes in the SQLDA according to the specification using the DESCRIBE statement (or the `sqldyndesc()` ESQL-LIB function). You can also initialize the column attributes yourself, if the information is available.
4. If you complete Step 2 and Step 3, allocate storage for each column of the specification according to the initialized column attributes in the SQLDA. (There are several techniques available to complete this step.)
5. Associate a cursor with the specification using the DECLARE CURSOR statement (or the `sqldynsetn()` ESQL-LIB function).
6. Execute the cursor specification using the OPEN statement (or `sqldynexec()`).
7. If you constructed your own SQLDA in Step 2 through Step 4, retrieve each row of data into your SQLDA using the dynamic FETCH statement (or the `sqldynftch()` ESQL-LIB function).

8. If you did not specify your own SQLDA, retrieve each row of data into a list of host language variables using the dynamic FETCH statement. This host language variable list must exactly match the number of columns in the specification result set.

   The preprocessor allocates and initializes its own SQLDA to reference this list of variables, deallocating and reallocating the SQLDA each time the FETCH statement is invoked.

9. If you use your own SQLDA in the previous steps, deallocate its structure and data storage. (There are several techniques available to complete this step.)

If you do not allocate and use your own SQLDA, your dynamic cursor request pays a significant penalty in memory management with each row retrieval. You must also be precise about matching the number of host language variables to the number of columns in your specification. However, you reduce your development cycle from eight to four steps, giving some of the most intricate and tedious code to the preprocessor. For more information on executing dynamic cursor specifications using your own SQLDA, see the "Executing Dynamic Cursor Specifications" section in this chapter.

For information on the dynamic ESQL statements used for cursor specifications, see the "Prepared Execution" section.

If you have dynamic parameter markers in the cursor specification, you also need to provide your own SQLDA or host language variable list to specify the values. However, dynamic parameter markers require closer attention. For more information, see the "Using Dynamic Parameter Markers" section in this chapter.

2.15.5 Prepared Execution for Non-cursor Specification

For dynamic SQL statements other than cursor specifications, follow these general steps to execute the request:

1. Prepare the SQL request using the PREPARE statement (or `sqldynprep()`), establishing an SQL request handle for the statement.

2. Execute the request using the EXECUTE statement (or `sqldynexec()`).

Many SQL statements other than cursor specifications can take dynamic parameter markers, to provide, for example, input value for columns and search criteria when executing INSERT and searched UPDATE statements. When preparing these statements, you can specify an input SQLDA or a host language variable list as part of the dynamic EXECUTE statement. This provides the parameter values that correspond to your input columns or search criteria. This is similar to specifying the output SQLDA or host language variable list as part of the dynamic FETCH statement to retrieve the corresponding rows of a cursor specification.

For information on using dynamic parameter markers in non-OPEN cursor statements, see the "Using Dynamic Parameter Markers" section in this chapter. For more information on the dynamic ESQL statements used for prepared execution, see the "Prepared Execution" section.

2.16 SQL Descriptor Area

The *SQL descriptor area* (SQLDA) is a dynamically allocated memory structure used to retrieve the result set from a dynamic cursor SELECT statement (where you do not know the identity of the columns ahead of time) or to specify the values for dynamic parameter markers. The SQLDA is composed of one instance of the SQLDA_T structure and multiple instances of the SQLDA_T structure as defined in the `proesql.h` header file in your ESQL installation directory.

Following is the exact structure of the SQLDA:

```
struct SQLVAR_T {
   char *DATA;      /* some kind of ptr to storage for holding the data */
   long *INDICATOR; /* ptr to storage for holding indicator variable */
   long  LENGTH;
   short TYPE;
   short SCALE;
   short PRECISION;
   short NULLABLE;  /* true if this column can be set to null */
   char  NAME[36];  /* column name */
};

struct SQLDA_T {
   short SQLN;     /* Number of SQLVAR_T instances allocated */
   short COLCNT;   /* number of columns actually in the current select */
   struct SQLVAR_T *SQLVAR[1]; /* ptr to 1st SQLVAR_T instance */
};
```

Each SQLDA_T structure in the SQLDA provides storage for one column, or in the case of array columns, one array element in a dynamic result set. When used for dynamic parameter markers, each SQLDA_T structure provides storage for the value of one parameter marker in a prepared statement.

When retrieving data into your ESQL application, the SQLDA contains, for each column in the SELECT result, the necessary information to describe each column to your application, and also a pointer to the actual data returned when a row is fetched. The SQLDA handles one row at a time. When specifying the values for dynamic parameter markers, the SQLDA contains, for each parameter marker, the information to describe the value to Progress, and also a pointer to the actual data that Progress uses.

2.16.1 Functions for Working with SQL Descriptor Areas

Table 2–11 lists a number of routines that are supplied in the ESQL-LIB interface that assist in defining and manipulating the SQLDA_T and SQLVAR_T structures.

Table 2–11: ESQL-LIB Routines

Routine	Description
`sqldynncols()`	Returns the number of columns in a cursor specification result set.
`sqldynnparms()`	Returns the number of parameter markers in a prepared statement.
`sqldsize()`	Returns the number of bytes required to allocate an SQLDA for a specified number of columns or parameter markers.
`sqlald()`	Allocates an SQLDA for a specified number of columns or parameter markers (also supporting the ALLOCATE DESCRIPTOR statement).
`sqldinit()`	Initializes an allocated SQLDA for a specified number of columns or parameter markers.
`sqldyndesc()`	Describes an SQLDA with the attribute information for all the columns that a prepared cursor specification specifies (also supporting the DESCRIBE statement).
`sqldyndescinp()`	Describes an SQLDA with the attribute information for all the parameter markers in a prepared statement (or cursor specification).
`sqlalloc()`	Allocates memory for the DATA or INDICATOR member of an SQLVAR_T structure.
`sqlfree()`	Deallocates memory for the DATA or INDICATOR member of an SQLVAR_T structure that `sqlalloc()` previously allocates.
`sqldld()`	Deallocates the SQLDA that `sqlald()` allocates (also supporting the DEALLOCATE DESCRIPTOR statement).

NOTE: You can also use your own memory management routines other than `sqlald()`, `sqlalloc()`, `sqlfree()`, and `sqldld()` to allocate and deallocate SQLDAs.

After you prepare an SQL request, you can use the sqldynncols() and sqldynnparms() functions to return the number of result set columns specified for a cursor specification or the number of parameter markers specified for any request, respectively.

The sqldsize() ESQL-LIB function determines the amount of space that needs to be allocated to hold the SQLDA information for a given number of columns in a result set or parameter markers in a prepared statement. This function is especially useful if you are using your own memory management routines.

To allocate the SQLDA for a specified number of columns or parameter markers, you can use the sqlald() ESQL-LIB function. This example shows function sqlald() allocating space for an SQLDA that can handle 10 result columns or parameter marker values:

```
if ((psqlda = sqlald((int) 10)) == (struct SQLDA_T *) 0)
  {
     printf("Couldn't alloc memory for SQLDA.\n");
     break;
  }
```

The sqlald() function allocates memory for one instance of the SQLDA_T structure and a specified number of instances of the SQLVAR_T structure. (In the previous example, 10 instances of SQLVAR_T were allocated.) It also initializes the allocated SQLDA_T and SQLVAR_T structures.

Once an SQLDA has been allocated, you can use the sqldinit() ESQL-LIB function to initialize the members of the SQLVAR_T structure and their pointers in the SQLDA_T structure. This function is especially useful if you use your own memory management routines.

Both the sqlald() and sqldinit() routines set the SQLN member of the SQLDA_T structure to the number (of columns) passed to it. In the previous example, psqlda->SQLN equals 10 after sqlald() returns.

After an SQLDA has been allocated, you can use the sqldyndesc() and sqldyndescrinp() functions to fill in the attribute information for the columns specified for a prepared cursor specification or the parameter markers specified for any prepared request, respectively.

The sqlald() routine allocates memory only for the SQLDA_T and SQLVAR_T structures and not for the memory needed to hold the column or parameter marker data. After you allocate and describe the SQLDA_T structure, allocate memory for the column or parameter marker data to hold the values or set the DATA and INDICATOR members to reference the memory. You can also use the sqlalloc() routine to allocate memory for column data.

Call the deallocation routines `sqldld()` and `sqlfree()` in the reverse order from the allocation routines. If you use the `sqlalloc()` routine to allocate memory for the column data, call `sqlfree()` to deallocate this memory prior to freeing the SQLDA_T structure, and if you use the `sqlald()` routine to allocate the SQLDA_T structure, then you must call `sqldld()` to free the memory when you are done with a cursor or done executing prepared statements that contain parameter markers.

This example deallocates memory with the `sqldld()` routine:

```
sqldld(psqlda);
```

Figure 2–3 shows a memory map of the SQLDA structure after you have allocated memory with `sqlald()` and `sqlalloc()`.

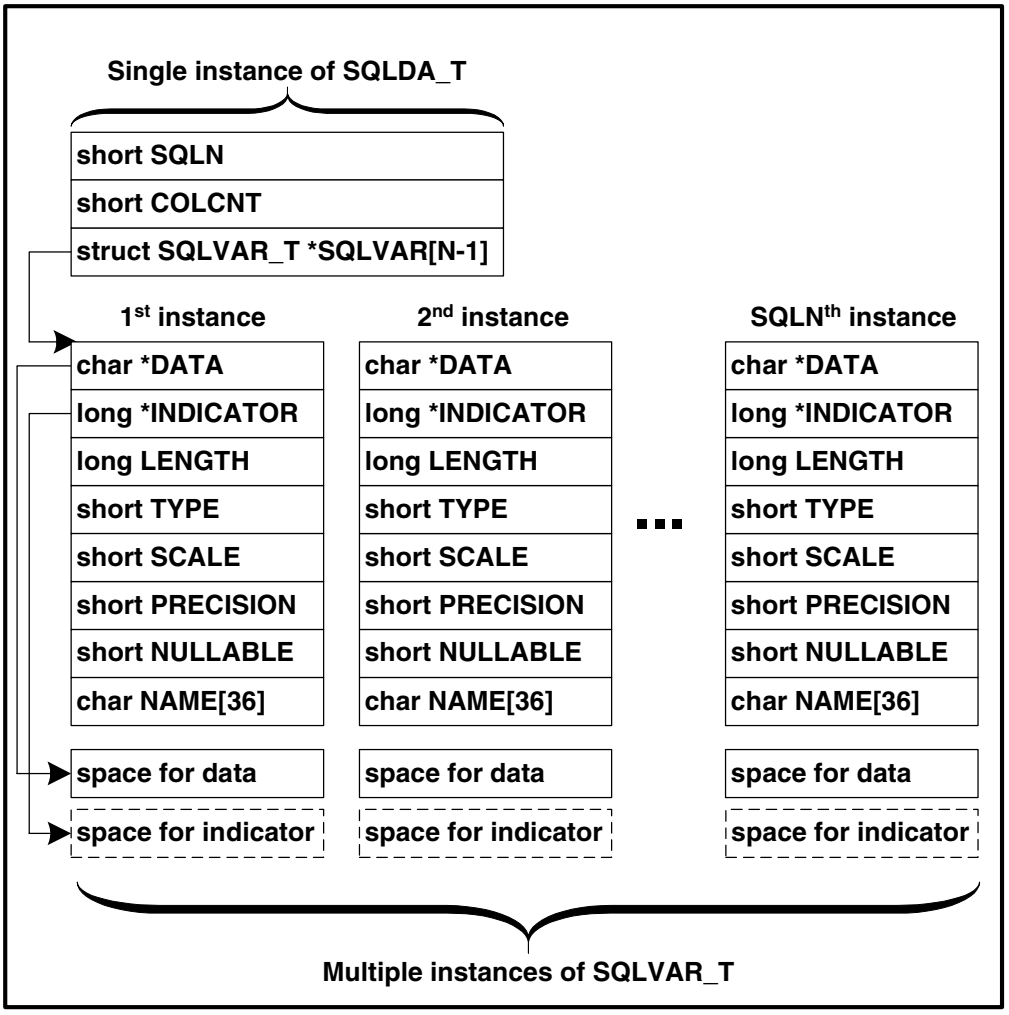

Figure 2–3: SQLDA_T Memory Map After Allocation

If you use the `sqlald()` routine, Progress allocates the instances of SQLVAR_T contiguously, directly following the SQLDA_T structure. The `sqldinit()` routine presumes at SQLVAR_T directly follows the SQLDA_T structure. Both the `sqlald()` and `sqldinit()` routines initialize the array of SQLVAR_T pointers to point to the SQLVAR_T structures.

2.16.2 SQLDA_T Structure Components

Table 2–12 describes the components of the SQLDA_T structure.

Table 2–12: SQLDA_T Structure Components

Component	Description
SQLN	Holds the number of occurrences of SQLVAR_T allocated for this descriptor.
COLCNT	Holds the actual number of SQLVAR_T structures that contain valid column or parameter marker data (the actual number of SQLVAR_T structures (that SQLN defines) might be greater than the number of columns in the result set or the number of parameter markers in the prepared statement). The COLCNT must match either of those values exactly.
SQLVAR[1]	This is an array of pointers to the SQLVAR_T structures. The SQLN member of the SQLDA_T structure determines the number of elements in this array.

2.16.3 SQLVAR_T Structure Components

Table 2–13 describes the components of the SQLVAR_T structure.

Table 2–13: SQLVAR_T Structure Components *(1 of 3)*

Component	Description
DATA	Pointer to the storage in which the value that a dynamic FETCH returns is to be stored, or where the value to be supplied for a parameter marker might be located. The program must set it before the first FETCH or prior to executing a prepared statement. Align the storage for the data type of the values being retrieved according to the rules of your machine architecture. Otherwise, a memory violation might occur.
INDICATOR	Pointer to a host language variable of type long, to indicate whether a fetched value is NULL (or has been truncated) or to indicate that a value for a parameter marker is to assume the NULL value. On a FETCH, the location to which INDICATOR refers is set to the value -1 if the column is NULL, and the total length of the column if the target value (as defined in the SQLDA) is too small for a character value. If INDICATOR returns -1, the value of DATA is undefined and should be ignored. Otherwise INDICATOR is set to 0. INDICATOR is unused until a FETCH is executed. The host program must set the pointer. If it is not set, and the value of the column is NULL, the same error code is returned in SQLCODE as would be returned if an ordinary static FETCH did not provide an indicator. On an EXECUTE, the ESQL-LIB interface examines the location to which INDICATOR refers in order to determine whether the value to be supplied for a parameter marker is NULL. If the location to which INDICATOR refers is set to the value of -1, then the NULL value is used as the value of the parameter marker. If you are not using the INDICATOR field, ensure that INDICATOR is set to 0. Otherwise, unpredictable results may occur.

Table 2–13: SQLVAR_T Structure Components *(2 of 3)*

Component	Description
LENGTH	When fetching data, the declared length for an SQL CHARACTER column (a column in a table created through a CREATE TABLE statement). For Progress CHARACTER fields, which have no declared length, LENGTH is the field length the Progress display format indicates. For example, a format of "X(20)" would yield a LENGTH of 20; a format of "(999) 999-9999" would yield a LENGTH of 10 (filler display characters are not counted, because the value is returned to the host program unformatted). The LENGTH does not include the null terminator required for the target variable. LENGTH is undefined for other data types and for all expressions. For parameter marker values, LENGTH is the length in bytes of an SQL CHARACTER value.
TYPE	Numeric code indicating the declared data type of the column: SQLDACHAR (1) for CHARACTER SQLDANUM (2) for NUMERIC SQLDADE (3) for DECIMAL SQLDAINT (4) for INTEGER SQLDASML (5) for SMALLINT SQLDAFLT (6) for FLOAT SQLDAREAL (7) for REAL SQLDADBL (8) for DOUBLE PRECISION SQLDADATE (-1) for Progress DATE SQLDALOG (-2) for Progress LOGICAL
SCALE	When fetching data, SCALE is set to 0 for INTEGER, SMALLINT, FLOAT, REAL, and DOUBLE PRECISION, and the declared scale for SQL DECIMAL or NUMERIC columns (for example, a declaration in a CREATE TABLE statement of DECIMAL(5,2) means a scale of 2). For Progress DECIMAL fields, SCALE is the value assigned to “Decimals” in the dictionary. SCALE is undefined for other data types and for derived columns (expressions). Any results that may be returned for SCALE for derived columns are unreliable and should not be used by the application. When supplying values for parameter markers, SCALE is ignored. For DECIMAL and NUMERIC values, the SCALE factor is obtained by scanning the C character string for the decimal separator.

Table 2–13: SQLVAR_T Structure Components *(3 of 3)*

Component	Description
PRECISION	When fetching data, the declared precision for SQL DECIMAL, NUMERIC and FLOAT columns. (For example, for DECIMAL(5,2) the precision is 5.) For Progress DECIMAL fields, *PRECISION* is calculated from the Progress display format, disregarding display fill characters. For example, a format of "->>,>>>,>>9.99" yields a PRECISION of 10. PRECISION is undefined for other data types, and for derived columns. Any results that may be returned for PRECISION for derived columns are unreliable and should not be used by the application. For decimal and numeric types, set PRECISION to 3 less than the length of the data, including the NULL character. For example, "-123.4" must set PRECISION to 4 with 7 bytes allocated.
NULLABLE	Is set to 1 if the column can have NULL values and 0 otherwise. It defaults to 0 for derived columns. This value is ignored when supplying values for parameter markers.
NAME	Holds the column name. This is undefined for derived columns. The first 33 bytes of NAME contain the column name (up to 32 bytes) and a NULL (\0) terminator. The NULL terminator is present at the end of the column name. For array columns, 3 to 6 bytes can be used by the subscript, depending on the maximum value of the extent (2 bytes + number of digits in subscript value). Thus, the array column name can be from 26 to 29 bytes, depending on this maximum extent value. Any name that is too long for the extent value generates an error. The last 3 bytes in NAME are reserved for internal ESQL use. This member is ignored when supplying values for parameter markers.

2.16.4 SQLDA Data Type Management

Internally, Progress does not differentiate between INTEGER and SMALLINT, between NUMERIC and DECIMAL, or between FLOAT and REAL. When retrieving column attribute information, all columns are described in terms of the following types:

- CHARACTER
- DECIMAL

- INTEGER
- DATE
- LOGICAL

Expressions are typed as CHARACTER, INTEGER (COUNT function), DECIMAL (all other numeric expressions), DATE, or LOGICAL. The Progress RECID function is INTEGER, and CAN-DO is LOGICAL.

Between the DESCRIBE statement (`sqldyndesc()` function) and the FETCH statement (`sqldynftch()` function) for cursor specifications, the host program can change the TYPE to indicate the actual target data type for a column. Likewise, between the `sqldyndescinp()` function and EXECUTE or OPEN statement (`sqldynexec()` function) for all dynamic statements using parameter markers, the host program can change the TYPE to indicate the actual source data type for a parameter marker. ESQL performs the necessary data type conversions. These types are specified in the preceding table and in `proesql.h`.

For numeric values, ESQL assumes that the target is long for INTEGER columns or parameter markers, and char for DECIMAL columns or parameter markers, unless the program changes the TYPE code before the FETCH, EXECUTE, or OPEN statement. DOUBLE PRECISION values can be fetched into a C double (or supplied from a C double) by setting the TYPE member of the SQLVAR_T structure to a value of SQLDABL.

NOTE: In previous versions of Progress, decimal data returns by default to the application as a floating point number. In effect, this converts an exact numeric value to an approximate numeric value, thereby potentially losing precision. In Version 7 and later, ESQL SQLDANUM and SQLDADEC are returned by default as C character arrays. It is up to the application to convert the values to the data types appropriate to the application, or to fetch into an SQLDADBY, SQLDAREAL, or SQLDAFLT.

Progress provides a variety of conversions between Progress/SQL and C data types in dynamic ESQL. For more information, see Appendix B, “ESQL Data Type Compatibilities.”

2.17 Executing Dynamic Cursor Specifications

Unlike static Embedded SQL, when you perform a cursor SELECT with dynamic ESQL, you cannot predict in advance the number of columns in the result, nor the data types of these columns. Instead, your application must have some way of handling and storing the results dynamically at run time.

Therefore, ESQL support for cursor SELECT statements in dynamic ESQL includes the following:

- Data structures (SQLDA) and ESQL functions that handle the column data results.
- Dynamic ESQL PREPARE and DESCRIBE statements to compile ESQL statements and initialize the SQLDA structures.
- Dynamic ESQL DECLARE CURSOR, OPEN, and FETCH statements to open a cursor and retrieve results.

NOTE: Multiple simultaneous cursors are supported for both static and dynamic ESQL.

This section provides a detailed description of the typical steps for executing a dynamic cursor SELECT statement using Progress/ESQL. The examples shown are taken, in part, from the dynamic ESQL demo application (`dyndemo.cc`) provided with your ESQL installation.

NOTE: In dynamic ESQL, make all single-row statements using cursor selects. Dynamic ESQL does not support the singleton SELECT (SELECT...INTO statement).

2.17.1 Dynamic Cursor SELECT Using Progress/ESQL

These are typical steps you take to execute a dynamic cursor SELECT using Progress/ESQL:

1 ♦ Declare a pointer to an instance of the ESQL SQLDA_T structure, as shown in the following example:

```
struct SQLDA_T *psqlda=(struct SQLDA_T *) 0;
```

For detailed information on the SQLDA, see the "SQL Descriptor Area" section.

2 ♦ Prepare the SQL statement as shown in this code fragment:

```
char *prequest = "SELECT * FROM customer FOR READ ONLY";
EXEC SQL PREPARE DEMOSTMT FROM prequest;
```

NOTE: You specify any lock criteria (FOR READ ONLY or FOR UPDATE) directly in the request string, along with the cursor specification. DEMOSTMT is an arbitrary statement identifier, like a cursor name.

The variable prequest is a pointer to the request string. It ultimately maps to a request handle in the C code.

NOTE: The scope of a statement identifier (used, for example, in PREPARE, DESCRIBE and DECLARE CURSOR statements) extends to the entire ESQL source file. In the example, this means that any reference to this statement identifier in any function within the file references the prepared statement in request.

3 ♦ Determine the number of columns in the result set by first calling the `sqlgetrqhd2()` ESQL-LIB function to obtain the request handle for the prepared cursor SELECT, and then calling the `sqldynncols()` function to return the number of columns:

```
EXEC SQL BEGIN DECLARE SECTION;
int    ncols;
EXEC SQL END DECLARE SECTION;
SQLRQHDL *shandle;
            .
            .
            .
sqlgetrqhdl("DEMOSTMT", &shandle);
sqldynncols(shandle, &ncols);
```

You can also select a number of columns known to be greater or equal to the columns in the result set. However, the following steps assume you use these ESQL-LIB functions to return the exact number in the host language variable ncols.

4 ♦ Allocate storage for the SQLDA_T and SQLVAR_T structures using your own memory allocation routines, the `sqlald()` ESQL-LIB function, or the ALLOCATE DESCRIPTOR statement that this function supports. The following example shows function `sqlald()` allocating space for structures that can handle `ncols` result columns as determined in Step 3 (The equivalent ALLOCATE DESCRIPTOR statement is shown in the comment.):

```
/* EXEC SQL ALLOCATE DESCRIPTOR psqlda WITH MAX :ncols; */

psqlda = sqlald((int) ncols);
if (psqlda == (struct SQLDA_T *) 0)
{
    printf("Couldn't alloc memory for SQLDA.\n");
    break;
}
```

5 ♦ Initialize each SQLVAR_T structure with the attributes of the columns in the result set, including data type and size information. You can do this by executing the DESCRIBE statement, calling the `sqldyndesc()` ESQL-LIB function directly, or providing the

information from your knowledge of the table. This example executes the DESCRIBE statement:

```
EXEC SQL DESCRIBE DEMOSTMT USING SQL DESCRIPTOR psqlda;
```

6 ♦ Based on the data type and size of each column, use your own memory allocation routines or the `sqlalloc()` ESQL-LIB function to allocate storage for the data in the columns. As described in the previous section, the data type for each column is contained in the TYPE members of the SQLVAR_T structures (each column or array column element in the result has its own SQLVAR_T instance). The DATA member of the SQLVAR_T structure points to the allocated storage for each column.

Also, if you can set the data type to a NULL value (the NULLABLE member of the SQLVAR_T structure is non-zero (TRUE), also allocate storage (again using `sqlalloc()` or the equivalent) for the indicator variable (a long type) to which the INDICATOR member of the SQLVAR_T structure points.

NOTE: See the `alloc_disp_cols()` function in the `dyndemo.cc` example for a comprehensive example of this step. This function uses `sqllalloc()` to allocate storage for an arbitrary number of columns with arbitrary data types, and displays the names of the columns.

7 ♦ Declare the cursor for the SELECT using the DECLARE CURSOR statement, as shown in this example:

```
EXEC SQL DECLARE C1 CURSOR FOR DEMOSTMT;
```

DEMOSTMT is the statement identifier used in the PREPARE statement in Step 2. You do not specify lock criteria in the dynamic DECLARE CURSOR statement, as you do in the static DECLARE CURSOR statement. Instead, you place lock criteria directly in the request string along with the cursor specification (see Step 2).

8 ♦ Open the cursor you just declared, as shown in this example:

```
EXEC SQL OPEN C1;
```

When you open a cursor, the SELECT statement you defined in your PREPARE statement (in Step 2) executes.

NOTE: If the SELECT statement to which the cursor name refers contains dynamic parameter markers, values for the parameter markers are supplied in the OPEN statement. For more information, see the "Using Dynamic Parameter Markers" section in this chapter.

9 ♦ Use the FETCH statement to access the results that the cursor SELECT generates, as shown in the following example:

```
EXEC SQL FETCH C1 INTO SQL DESCRIPTOR psqlda;
```

The FETCH statement retrieves one row of data into the data areas allocated in Step 6. You must access these data areas to access the column data.

NOTE: See the `disp_data()` function in the installed `dyndemo.cc` example for a comprehensive example of this step. This function reads and displays the data for an arbitrary number of columns with arbitrary data types.

10 ♦ When you are done with the SELECT statement, use the CLOSE statement to close the cursor, as shown in the following example:

```
EXEC SQL CLOSE C1;
```

11 ♦ Deallocate the storage you allocated in Step 4 and Step 6. Use `sqlfree()` to free any memory allocated using `sqlalloc()`. The `sqldld()` does not call `sqlfree()` to do this for you.

CAUTION: If you call `sqldld()` before deallocating memory previously allocated with `sqlalloc()`, the pointers to that memory are lost and the memory cannot be recovered until your application terminates.

See the `dealloc_cols()` function in the installed `dyndemo.cc` example for a comprehensive example of this step.

The following section describes how to use dynamic parameter markers in the cursor SELECT statement and any other dynamic ESQL request, using prepared execution. For more information on the dynamic ESQL statements used for prepared execution, see the "Dynamic ESQL Statements for Prepared Execution" section in this chapter.

2.18 Using Dynamic Parameter Markers

Dynamic parameter markers allow an ESQL application to prepare a dynamic ESQL statement once and execute it multiple times, with different values for the parameter markers each time the request executes. Therefore you bypass the extra compilation overhead for each new set of values. Dynamic parameter markers are represented with the question mark (?).

For example, consider the following dynamic ESQL request string:

```
char *request;
prequest = "SELECT name FROM customer WHERE cust\-num > ?";
```

To execute such a request, you have to prepare it, declare a cursor for it, and execute it like any cursor SELECT. However, you also have to specify a host language variable list or SQL descriptor area to provide new input values for the parameter markers each time you execute the request.

The following sections describe where in SQL you can use parameter markers and how to use them in your application.

2.18.1 Parameter Marker Syntax

Dynamic parameter markers can appear in any of the following parts of SQL statements:

- Expressions contained within a `WHERE` clause:

```
char *request;
prequest = "SELECT name FROM customer WHERE name < ?";
```

- Predicates, including:
  - BETWEEN
  - LIKE
  - BEGINS
  - NULL tests

- ALL or ANY
- EXISTS

```
char *request;
prequest = "SELECT name FROM customer WHERE name BEGINS ? ESCAPE ?";
```

- Expressions contained within a HAVING clause:

```
char *request;
prequest =  "SELECT name, cust\-num FROM order
            GROUP BY name HAVING order\-num < ?";
```

- The following functions:
  - AVG
  - MAX
  - MIN
  - SUM
  - COUNT
  - RECID
  - CAN-DO

```
char *request;
prequest =  "SELECT name, cust\-num FROM customer
          GROUP BY name HAVING AVG(max\-credit + ?) = 150";
```

- The value list of an INSERT statement:

```
char *request;
prequest =  "INSERT INTO customer (name, cust\-num)
          VALUES (?, 50 + ?)";
```

- As part of a SET value in an UPDATE statement:

```
char *prequest;
prequest =  "UPDATE order SET name = ?, cust\-num = 50 + ?
          WHERE order\-num = ?";
```

Currently, dynamic parameter markers cannot appear:

- In selection lists of SELECT statements.
- In array references. Although ESQL allows array references that can contain expressions, array references are not part of any SQL standard.
- In the expressions of GROUP BY and ORDER BY clauses. Although ESQL allows expressions in these clauses, the ANSI standard does not.
- Surrounding the IN predicate. (In the expression before an IN predicate and in the value list following it.)

2.18.2 Determining Parameter Marker Data Types

When you use a parameter marker, ESQL determines a default data type for the parameter marker from the context in which the parameter marker appears. In general, ESQL assumes for any dynamic parameter marker that is an operand of a binary arithmetic or comparison operator, that its data type is the same as the other operand of the operator. For example, the dynamic parameter marker in this expression has the same data type as the 5, INTEGER (SQLDAINT):

```
... WHERE cust\-num = 5 + ? ...
```

Table 2–14 lists the default parameter marker data types in different statement locations.

Table 2–14: Default Parameter Marker Data Types *(1 of 2)*

Location of Dynamic Parameter Marker	Default Data Type
An operand of a binary arithmetic or comparison operator.	Same as the other operand.
BETWEEN — first operand.	Same as the second operand.

Table 2–14: Default Parameter Marker Data Types *(2 of 2)*

Location of Dynamic Parameter Marker	Default Data Type
BETWEEN — second or third operand.	Same as the first operand.
IN — expression.	Same as either the first value or the result column of the subquery.
IN — value.	Same as the expression.
Expression in quantified predicate.	Same as the result column of the subquery.
INSERT — column value.	Same as the inserted column.
LIKE — pattern value.	CHARACTER (SQLDACHAR).
LIKE — ESCAPE character.	CHARACTER (SQLDACHAR).
UPDATE — column value.	Same as the updated column.

If you cannot determine the data type of a dynamic parameter marker based on the rules identified in this table, the dynamic parameter marker is invalid. For example, the following expression is invalid:

```
... WHERE cust\-num = ? + ? ...
```

2.18.3 Writing an Application with Parameter Markers

All applications that support dynamic ESQL statements carry out these basic operations.

1. Prepare a dynamic ESQL statement.
2. If the statement is a cursor request, declare a cursor and obtain column attribute information and storage.
3. Execute the statement, possibly multiple times.

Applications perform these same operations for dynamic ESQL statements with dynamic parameter marker support. There are only a few additional steps required to support dynamic parameter markers.

The following sections provide an overview of the steps you take to write an application with dynamic parameter markers. The techniques are different depending on whether you use Progress/ESQL or write directly to the ESQL-LIB interface.

2.18.4 Using Progress/ESQL and Parameter Markers

Currently, there is no SQL preprocessor support for dynamic parameter markers in Progress/ESQL statements. Consequently, if you choose to use the preprocessor exclusively, the only types of dynamic ESQL with parameter markers that you can include in an application are those statements in which you know the exact number and data types of dynamic parameter markers.

However, if you include additional calls to ESQL-LIB, you can use parameter markers in a more flexible manner. Using the `sqlgetrqhdl()` function, your ESQL application can gain access to the underlying ESQL-LIB interface and to parameter marker information that is unavailable through the Progress/ESQL syntax alone. See the sample program `dprmdemo.cc` which uses a combination of Progress/ESQL and ESQL-LIB to support parameter markers. For information on preparing and executing dynamic ESQL with parameter markers using only ESQL-LIB, see the "Using ESQL-LIB and Parameter Markers" section.

Handling Parameter Markers with Progress/ESQL

These are the typical steps with Progress/ESQL to prepare and execute SQL statements that use dynamic parameter markers.

1 ♦ Ensure that you have declared a pointer to each SQLDA_T structure — at least one for parameter marker values and one for the result set of any cursor specification.

2 ♦ Prepare a statement using the PREPARE statement.

3 ♦ If the statement is a cursor specification, create a column SQLDA_T structure:

a) Allocate an SQLDA_T structure for the result set columns using the ALLOCATE DESCRIPTOR statement (or your own memory allocation routine). Be sure to specify a large enough number of columns to handle the cursor specification. For more information, see the "Executing Dynamic Cursor Specifications" section in this chapter.

b) Initialize the SQLDA_T structure with the column attribute information of the result set using the DESCRIBE statement.

c) For each column (SQLVAR_T structure), assign storage addresses (allocated, if necessary) to the DATA and INDICATOR members to hold the column result.

4 ♦ Call the `sqlgetrqhdl()` function to retrieve the SQL request handle associated with the SQL statement identifier of your prepared statement.

5 ♦ Call `sqldynnparms()` to retrieve the number of parameter markers in the SQL statement.

6 ♦ Allocate an SQLDA_T structure to hold the parameter marker values using the ALLOCATE DESCRIPTOR statement (or your own memory allocation routine).

7 ♦ Call `sqldyndescinp()` to initialize the parameter SQLDA_T structure with assumed data type information for each parameter marker in the prepared statement.

8 ♦ For each parameter marker (SQLVAR_T of the parameter SQLDA_T structure), assign storage addresses (allocated, if necessary) to the DATA and INDICATOR members to hold the parameter marker value. (Adjust the data type of the parameter marker, if necessary, to match the data type of the storage address.)

9 ♦ If the statement is a cursor specification, declare a cursor using the DECLARE CURSOR statement.

10 ♦ For each parameter marker (SQLVAR_T of the parameter SQLDA_T structure), assign the appropriate value(s) to the DATA and INDICATOR storage areas.

11 ♦ Execute the statement:

If the statement is a cursor specification:	If the statement is not a cursor specification:
Invoke the OPEN statement for the cursor, specifying the parameter SQLDA_T structure in the USING clause to pass parameter values to the statement. Execute the fetch loop to retrieve the rows using the dynamic ESQL FETCH statement.	Invoke the EXECUTE statement for the prepared statement identifier, specifying the parameter SQLDA_T structure in the USING clause to pass parameter values to the statement.

The USING clause of the OPEN and EXECUTE statement can specify a list of host language variable references (including indicator variables), instead of an SQL descriptor area. If you specify host language variable references in the USING clause, the SQL preprocessor declares its own SQL descriptor area to reference them.

In this case, skip steps 4 through 8, and in step 10, assign the parameter marker values to the specified host language variables. Ensure that your host language variable list is compatible in number and data type with the parameter markers in the prepared statement

or cursor specification. For more information, see the "Host Language Variables" and "Indicator Variables" sections in this chapter.

12 ♦ Repeat the statement as often as you want with new parameter values.

If the statement is a cursor specification, first close the existing query after your last fetch using the CLOSE statement, then repeat steps 9 through 11 as necessary to supply new values for the parameter markers.

If the statement is not a cursor specification, repeat steps 10 and 11 as necessary to supply new values for the parameter markers.

13 ♦ Deallocate any storage you allocated for DATA and INDICATOR members of SQLVAR_T structures and any SQLDA_T structures that you allocated using the DEALLOCATE DESCRIPTOR statement (or your own memory deallocation routine).

NOTE: If you prepare another statement using the statement identifier associated with an open cursor, the cursor is automatically closed and the corresponding statement and its memory resources are released; the memory resource for any SQLDA structures that your application uses are untouched.

2.18.5 Using ESQL-LIB and Parameter Markers

Using ESQL-LIB alone to implement your dynamic ESQL application with parameter markers provides more flexibility than it does without parameter markers. However, you must explicitly manage all ESQL error and status condition handling for each ESQL-LIB function. You must also declare and manage your own SQL request handles (the ESQL-LIB equivalent of dynamic ESQL statement identifiers). For more information on error handling, see the "Error Handling" section. For more information on SQL request handles, see Chapter 4, "ESQL-LIB Reference." See `clidemo.c` for an example of parameter marker support using only ESQL-LIB.

Handling Parameter Markers with ESQL-LIB

These are the typical steps with ESQL-LIB to prepare and execute SQL statements that use dynamic parameter markers.

1 ♦ Declare an SQL request handle for the statement you are about to prepare.

2 ♦ Prepare a statement by calling `sqldynprep()`.

3 ♦ If the statement is a cursor specification, create a column SQLDA_T structure:

a) Determine the number of the column that the cursor specification requires by calling the `sqldynncols()` function.

b) Allocate an SQLDA_T structure for the result set columns by calling the `sqlald()` function (or your own memory allocation routine).

c) Initialize the SQLDA_T structure with the column attribute information of the result set by calling the `sqldyndesc()` function.

d) For each column (SQLVAR_T structure), assign storage addresses (allocated, if necessary) to the DATA and INDICATOR members to hold the column result.

4 ♦ Call `sqldynnparms()` to retrieve the number of parameter markers in the SQL statement.

NOTE: You do not need to call `sqlgetrqhdl()`, as you would when using Progress/ESQL statements. You are already working with your own declared SQL request handle.

5 ♦ Allocate an SQLDA_T structure to hold the parameter marker values by calling the `sqlald()` function (or your own memory allocation routine).

6 ♦ Call `sqldyndescinp()` to initialize the parameter SQLDA_T structure with assumed data type information for each parameter marker in the prepared statement.

7 ♦ For each parameter marker (SQLVAR_T of the parameter SQLDA_T structure), assign storage addresses (allocated, if necessary) to the DATA and INDICATOR members to hold the parameter marker value. (Adjust the data type of the parameter marker, if necessary, to match the data type of the storage address.)

8 ♦ If the statement is a cursor specification and you are referencing the cursor elsewhere in your application (such as for a positioned UPDATE), set a cursor name by calling `sqldynsetn()`.

9 ♦ For each parameter marker (SQLVAR_T of the parameter SQLDA_T structure), assign the appropriate value(s) to the DATA and INDICATOR storage areas.

10 ♦ Execute the statement by calling the `sqldynexec()` function. Pass to `sqldynexec()` the address of the parameter SQLDA_T structure that you allocated in step 5 to supply the current values for parameter markers in the statement. If the statement is a cursor specification, execute the fetch loop to retrieve the rows by calling the `sqldynftch()` function. Pass to `sqldynftch()` the address of the column SQLDA_T structure that you allocated in step 3 to return the result set.

11 ♦ Repeat the statement as often as you want with new parameter values:

If the statement is a cursor specification:	If the statement is not a cursor specification:
Close the existing query after your last fetch by calling the `sqldynclose()` function with its input parameter set to the value of SQL_STMTCLOSE, then repeat steps 8 through 10 as necessary to supply new values for the parameter markers.	Repeat steps 9 and 10 as necessary to supply new values for the parameter markers.

12 ♦ Close any cursors you no longer plan to use by calling the `sqldynclose()` function with its input parameter set to the value of SQL_STMTDROP.

This permanently releases all memory associated with the cursor specification. It does not deallocate storage associated with any SQLDA_T structures.

NOTE: If you prepare another statement using the SQL request handle associated with an open cursor, the cursor is automatically closed and the corresponding statement and its memory resources are released.

13 ♦ Deallocate any storage you allocated for DATA and INDICATOR members of SQLVAR_T structures and any SQLDA_T structures that you allocated by calling the `sqldld()` function (or your own memory deallocation routine).

2.19 Dynamic ESQL Statements for Prepared Execution

The following sections describe the dynamic ESQL statements used for prepared execution. They appear in the order in which you typically use them.

2.19.1 Dynamic PREPARE Statement

The dynamic PREPARE statement associates an arbitrary SQL identifier with a character variable and compiles the SQL statement to which the character variable refers. The preprocessor generates a call to `sqldynprep()` for this statement.

Following is the syntax for the dynamic PREPARE statement:

SYNTAX

```
EXEC SQL PREPARE statement-id FROM request-string;
```

statement-id

An SQL identifier (limited to 18 characters) that identifies the prepared statement to other dynamic ESQL statements. The scope of a statement identifier (used also in the DESCRIBE, DECLARE CURSOR, and EXECUTE statements) extends to the entire ESQL source file. This means that any reference to this statement identifier in any function within the file refers to the same prepared statement.

request-string

A C character string (request string) that contains the SQL statement you want to prepare. This statement can be a cursor specification, INSERT, UPDATE (positioned or searched), DELETE (positioned or searched), or DDL statement. See the "Specifying SQL Statements" section for more information and additional restrictions on the request-string.

NOTE: Although you can prepare DDL statements, there is no advantage over using immediate execution. These statements do not accept dynamic parameter markers and the same DDL statement is typically not executed more than once in a session.

For an example of the dynamic ESQL PREPARE statement, see Step 2 in the "Executing Dynamic Cursor Specifications" section in this chapter.

2.19.2 Dynamic ALLOCATE DESCRIPTOR Statement

The dynamic ALLOCATE DESCRIPTOR statement provides a way to allocate memory for an SQLDA_T structure using Progress/ESQL. The preprocessor generates a call to `sqlald()` for this statement.

Following is the syntax for the dynamic ALLOCATE DESCRIPTOR statement:

SYNTAX

```
EXEC SQL ALLOCATE DESCRIPTOR psqlda [ WITH MAX :ncols ];
```

psqlda

The name of a pointer to an SQLDA_T structure.

ncols

The name of a host language integer variable that specifies the number of columns or parameter markers for which you want to allocate the structure. If you do not specify the WITH clause to specify this value, the value 10 is specified as the default.

For an example of an SQLDA_T structure allocation, see Step 4 in the "Executing Dynamic Cursor Specifications" section in this chapter.

2.19.3 Dynamic DESCRIBE Statement

The dynamic DESCRIBE statement associates a previously defined statement identifier with a previously declared SQLDA_T structure variable name. The preprocessor generates a call to `sqldyndesc()` for this statement.

Following is the syntax for the dynamic DESCRIBE statement:

SYNTAX

```
EXEC SQL DESCRIBE statement-id USING SQL DESCRIPTOR structure-name;
```

statement-id

A statement identifier previously defined in a PREPARE statement. The scope of a statement identifier (used also in the DECLARE CURSOR and EXECUTE statements) extends to the entire ESQL source file. Any reference to this statement identifier in any function within the file refers to the same prepared statement.

psqlda

The name of a pointer to a previously allocated SQLDA_T structure.

For an example of the dynamic ESQL DESCRIBE statement, see Step 5, and for an example of an SQLDA_T structure allocation, see Step 4 in the "Executing Dynamic Cursor Specifications" section in this chapter.

2.19.4 Dynamic DECLARE CURSOR Statement

The dynamic DECLARE CURSOR statement associates a cursor with a cursor specification that you previously prepared using a PREPARE statement. The preprocessor generates a call to `sqldynsetn()` for this statement. Multiple cursors are supported for both static and dynamic ESQL.

Following is the syntax for the dynamic DECLARE CURSOR statement:

SYNTAX

```
EXEC SQL DECLARE cursor-name CURSOR FOR statement-id ;
```

cursor-name

An SQL identifier (limited to 18 characters) that specifies the name of a cursor.

statement-id

A statement identifier associated with a cursor specification in a previous PREPARE statement. The scope of a statement identifier (used also in the DESCRIBE and EXECUTE statements) extends to the entire ESQL source file. This means that any reference to this statement identifier in any function within the file refers to the same cursor specification.

NOTE: Unlike the static DECLARE CURSOR statement, which includes the complete text of the cursor specification, you do not include lock criteria in a dynamic ESQL DECLARE CURSOR statement. For dynamic ESQL, you specify lock criteria in the SQL request string following the cursor specification.

For an example of the dynamic ESQL DECLARE CURSOR statement, see Step 7 in the "Executing Dynamic Cursor Specifications" section in this chapter.

2.19.5 Dynamic OPEN Statement

The dynamic OPEN statement opens the named cursor by executing the prepared SELECT statement. The preprocessor generates a call to `sqldynexec()` for this statement.

Following is the syntax for the dynamic OPEN statement:

SYNTAX

```
EXEC SQL OPEN cursor-name
 [ USING { SQL DESCRIPTOR structure-name | host-var-ref
 [ [ INDICATOR ] ind-var-ref ] , ... } ] ;
```

cursor-name

A cursor name that you previously defined in a DECLARE CURSOR statement.

structure-name

If the prepared SELECT statement to which the cursor name indirectly refers has dynamic parameter markers, the *structure-name* is the name of an SQLDA_T structure that contains the values to be supplied for those parameter markers.

host-var-ref

Host language variable references (including any indicator variables) are used to supply values for dynamic parameter markers, if any, in the prepared statement. These variables must appear in the SQL Declare section of your ESQL application. For more information, see the "Host Language Variables" and "Indicator Variables" sections in this chapter.

ind-var-ref

An indicator variable provides information about a host language variable that is not representable in the host language variable value itself. You declare indicator variables like host language variables in an SQL Declare section, and declare them as C numeric types. Indicator types can be long, short, sqldec, float, or double. For information on declaring host language variables, see the "Host Language Variables" section in this chapter.

When using the SQLDA_T structure, the OPEN statement expects the DATA member of the SQLVAR_T structure for each column to point to the target storage for the data value returned. It also expects the INDICATOR member to point to a long (which you can set to 0 or -1), if the program allocates storage for INDICATOR.

CAUTION: Any bad pointer values in DATA or INDICATOR members can lead to unpredictable application behavior.

For an example of the dynamic ESQL OPEN statement, see Step 8 in the "Executing Dynamic Cursor Specifications" section in this chapter.

2.19.6 Dynamic EXECUTE Statement

The dynamic EXECUTE statement executes any SQL statements other than cursor specifications that you prepare using the PREPARE statement. The preprocessor generates a call to `sqldynexec()` for this statement.

Following is the syntax for the dynamic EXECUTE statement:

SYNTAX

```
EXEC SQL EXECUTE statement-name
 [ USING { SQL DESCRIPTOR structure-name | host-var-ref
 [ [ INDICATOR ] ind-var-ref ] , ... } ] ;
```

statement-name

A statement identifier previously defined in a PREPARE statement. The scope of a statement identifier (used also in the DESCRIBE and DECLARE CURSOR statements) extends to the entire ESQL source file. Any reference to this statement identifier in any function within the file references the same prepared statement.

structure-name

If the prepared statement contains dynamic parameter markers, the *structure-name* is the name of an SQLDA_T structure that contains the values to be supplied for those parameter markers.

host-var-ref

Host language variable references (including any indicator variables) are used to supply values for dynamic parameter markers, if any, in the prepared statement. These variables must appear in the SQL Declare section of your ESQL application. For more information, see the "Host Language Variables" and "Indicator Variables" sections in this chapter.

ind-var-ref

An indicator variable provides information about a host language variable that is not representable in the host language variable value itself. You declare indicator variables like host language variables in an SQL Declare section, and declare them as C numeric types. Indicator types can be long, short, sqldec, float, or double. For information on declaring host language variables, see the "Host Language Variables" section in this chapter.

When using the SQLDA_T structure, the EXECUTE statement expects the DATA member of the SQLVAR_T structure for each column to point to the target storage for the data value returned. It also expects the INDICATOR member to point to a long (which you can set to 0 or -1), if the program allocates storage for INDICATOR.

CAUTION: Any bad pointer values in DATA or INDICATOR members can lead to unpredictable application behavior.

2.19.7 Dynamic FETCH Statement

The dynamic FETCH statement retrieves the rows of a dynamic cursor similar to the way a static FETCH statement retrieves the rows of a static cursor. The preprocessor generates a call to `sqldynftch()` for this statement.

Following is the syntax for the dynamic FETCH statement:

SYNTAX

```
EXEC SQL FETCH cursor_name
 { USING SQL DESCRIPTOR structure-name | INTO host-var-ref
  [ [ INDICATOR ] ind-var-ref ] , ... } ;
```

cursor-name

A cursor name previously defined in a DECLARE CURSOR statement.

structure-name

A previously declared SQLDA_T structure variable name used to retrieve the values for the row being fetched.

host-var-ref

Host language variable references (including any indicator variables) are used to retrieve column data. These variables must appear in the SQL Declare section of your ESQL application. For more information, see the "Host Language Variables" and "Indicator Variables" sections in this chapter.

ind-var-ref

An indicator variable provides information about a host language variable that is not representable in the host language variable value itself. You declare indicator variables like host language variables in an SQL Declare section, and declare them as C numeric types. Indicator types can be long, short, sqldec, float, or double. For information on declaring host language variables, see the "Host Language Variables" section in this chapter.

When using the SQLDA_T structure, the FETCH statement expects the DATA member of the SQLVAR_T structure for each column to point to the target storage for the data value returned. It also expects the INDICATOR member to point to a long (which you can set to 0 or -1), if the program allocates storage for INDICATOR.

CAUTION: Any bad pointer values in DATA or INDICATOR members can lead to unpredictable application behavior.

When using host language variable references, the preprocessor automatically allocates and initializes an SQLDA_T structure to refer to the specified host language and indicator variables for the fetch.

The ESQL-LIB interface returns data to your application based on the values in the SQLDA_T structure at the time of the fetch, and it assigns the row values to the respective targets based on the addresses in DATA and INDICATOR, and the values for TYPE and LENGTH at that time.

Table 2–15 lists the assumed data types for target storage associated with the SQL data types specified in the SQLDA_T structure for a FETCH statement.

Table 2–15: Data Types for Target Storage

Data Type	Target
CHARACTER	Type char, and LENGTH length-1 (allowing for a null terminator). LENGTH is ignored for all other data types.
INTEGER	C long
SMALLINT	C short
DECIMAL, or NUMERIC	ESQL sqldec
REAL, or FLOAT	C float
DOUBLE PRECISION	C double
LOGICAL	ESQL sqlbool
DATE	ESQL sqldate

The Progress implementation of dynamic ESQL allows you to change the values of TYPE, SCALE, PRECISION, and LENGTH between a DESCRIBE and corresponding FETCH statement. This enables you to have the ESQL-LIB interface coerce the value into a different data type from the defaults listed in Table 2–15.

For more information on data type conversion, see Appendix B, "ESQL Data Type Compatibilities." For an example of the dynamic ESQL FETCH statement, see Step 9 in the "Executing Dynamic Cursor Specifications" section in this chapter.

2.19.8 Dynamic CLOSE Cursor Statement

The dynamic CLOSE statement closes a dynamic cursor much like a static cursor, except that the prepared statement is not immediately discarded. Closing the cursor discards all pending rows for the request, but allows you to reopen the cursor without preparing the cursor SELECT associated with the cursor. This allows you to re-execute the cursor specification, possibly with new values for any parameter markers, without recompiling the request.

The request is finally dropped when you prepare another (possibly identical) request using the same statement identifier (or request handle), or a transaction commit or transaction rollback occurs. The preprocessor generates a call to `sqldynclose()` for this statement.

Following is the syntax for the dynamic CLOSE statement:

SYNTAX

```
EXEC SQL CLOSE cursor-name;
```

cursor-name

A cursor name previously defined in a DECLARE CURSOR statement.

NOTE: The SQLDA is not automatically deallocated or cleared when the cursor is closed.

For an example of the dynamic ESQL CLOSE statement, see Step 10 in the "Executing Dynamic Cursor Specifications" section in this chapter.

2.19.9 Dynamic DEALLOCATE DESCRIPTOR Statement

The dynamic DEALLOCATE DESCRIPTOR statement provides a way of using Progress/ESQL to deallocate the memory for an SQL descriptor area (SQLDA_T structure) that you previously allocated using the ALLOCATE DESCRIPTOR statement (or the `sqlald()` ESQL-LIB function). The preprocessor generates a call to `sqldld()` for this statement.

Following is the syntax for the dynamic DEALLOCATE DESCRIPTOR statement:

SYNTAX

```
EXEC SQL DEALLOCATE DESCRIPTOR psqlda;
```

psqlda

The name of a pointer to an SQLDA_T structure that an ALLOCATE DESCRIPTOR statement or `sqlald()` function previously allocates.

CAUTION: Deallocate all allocated memory to which SQLVAR_T structures refer in the SQL descriptor area before you deallocate the descriptor area. Otherwise, the memory pointers (DATA and INDICATOR) are lost and the memory cannot be freed.

Do not deallocate the same SQL descriptor area more than once. This can cause a memory violation. Do not attempt to invoke the DEALLOCATE DESCRIPTOR statement with any value for psqlda other than what a previous ALLOCATE DESCRIPTOR statement or call to `sqlald()` returns. This causes unpredictable behavior.

3

Building and Running ESQL Applications

After coding your ESQL application, you are ready to determine whether to build a large or small client executable, build the executable, and then run it.

This chapter describes:

- Issues in choosing an ESQL configuration
- Overview of building an ESQL executable
- Preprocessing ESQL source files
- Compiling preprocessed source files
- Linking ESQL executables
- Running ESQL applications
- Running the ESQL demo applications

3.1 Choosing an ESQL Configuration

Progress allows you to configure your ESQL application using either the large client or small client model. Your choice depends on your application and the resources in your environment.

3.1.1 ESQL Large Client Model

The large client model provides a single ESQL client executable that contains both your application code and a complete Progress database client environment. The large client executable accesses the database like any Progress database client, through database server processes or as a self-service client (dotted lines). Figure 3–1 shows the process layering in the ESQL large client model.

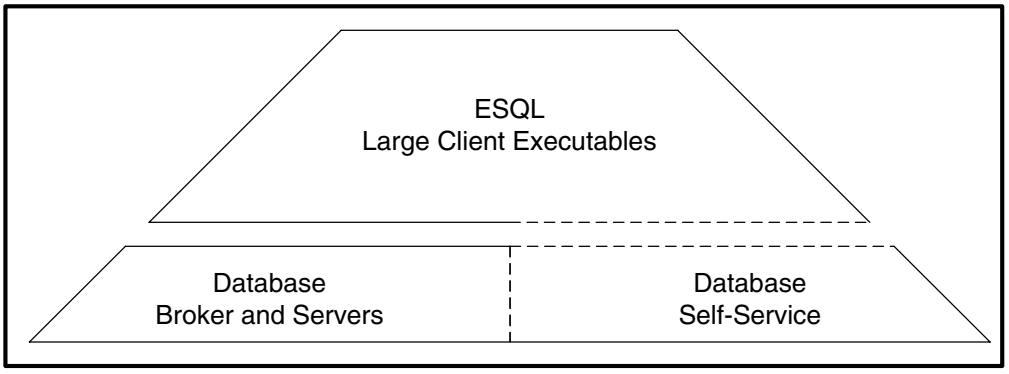

Figure 3–1: Large Client Process Model

In the large client model, each instance of your ESQL application is linked with Progress in a single executable image that accesses the database directly. This allows you to run as a self-service Progress client or a remote multi-user Progress client.

Although the large client model allows you to run ESQL applications as either local self-service database clients or remote clients, the model is most appropriate in a single-user self-service environment where no network is involved. You might want to run large clients in a multi-user self-service environment, especially when:

- Your ESQL application size is relatively small.
- The application has no signal, global name, or memory conflicts with Progress internals.

However, you can still run a remote large client with a network connection to the database server.

Figure 3–2 shows typical local large client configurations.

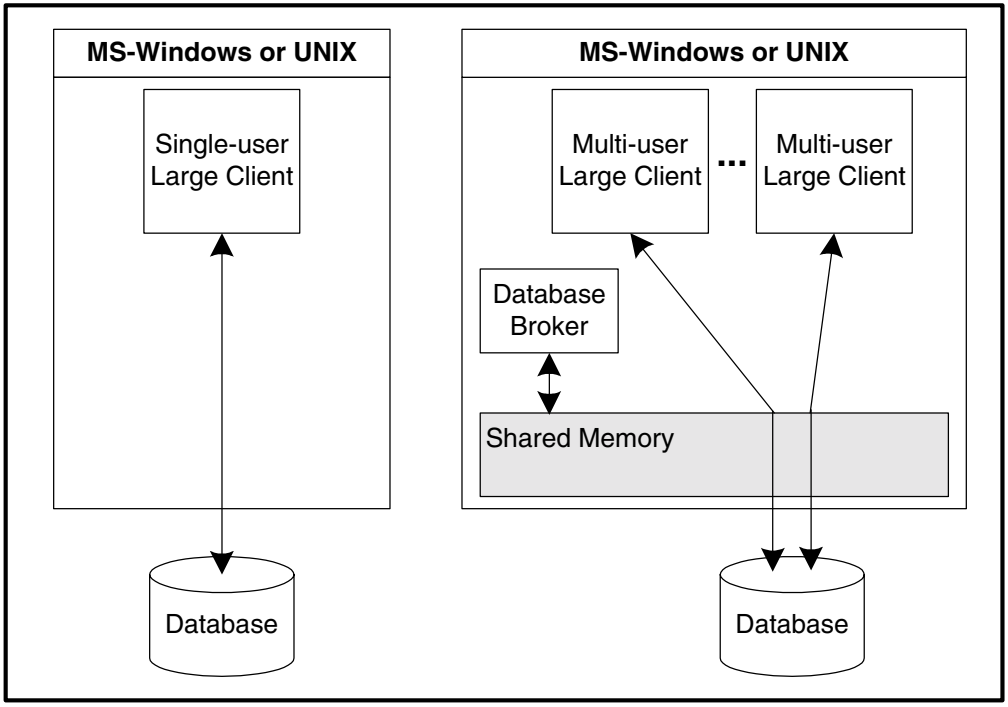

Figure 3–2: Typical Local Large Client Configurations

3.1.2 ESQL Small Client Model

The ESQL small client model separates the large client executable into two components:

- The first component is the small client executable that contains your ESQL application and Progress client communications modules. On Windows, the ESQL application is linked to `esql32.lib`; on UNIX, the application is linked directly to client modules.
- The second component is the Open Interface Driver (OID) that runs as a Progress database client on behalf of your ESQL application. The Open Interface Broker (OIB) brokers the connections to the OID. The OID accesses the database like any Progress database client, through database server processes or as a self-service client.

The OID can run on a machine that is either remote or local to the database server machine. In response to small client requests, the OIB starts OID processes — one server for each small client that logs into Progress.

An ESQL small client application can run on Windows or UNIX workstations, either remotely or locally on the OID machine. Figure 3–3 shows the process layering in the ESQL small client model.

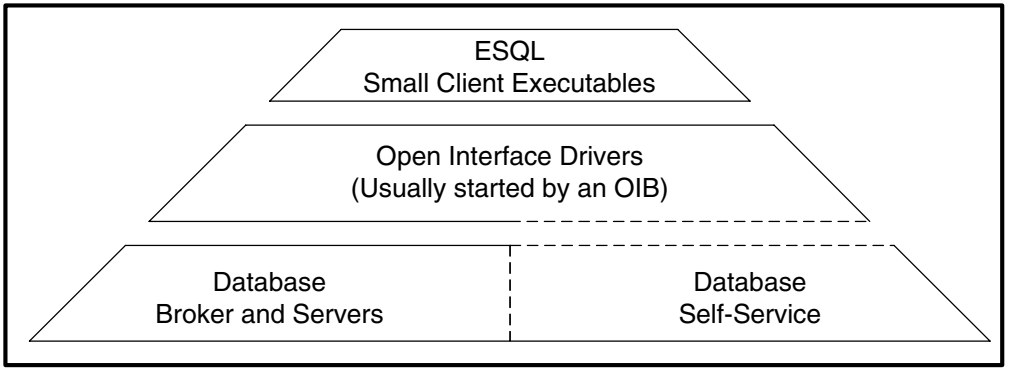

Figure 3–3: Small Client Process Model

Network Configurations

Typically, an ESQL small client application runs remotely from its OID, and the OID runs as a self-service database client on the database server machine. As a general rule, having the OID on the same machine as the database server is the optimal configuration. The OID interprets and returns result sets for all ESQL database requests from the small client. The small client receives only the final results of these requests across the network from its OID, minimizing network traffic. (To ensure this advantage, the OID, database server, and database should reside on the same machine.)

If you use a remote large client instead, the database server sends all database tables that the Progress/SQL interpreter requires across the network to the large client, substantially increasing network load.

Unlike the large client model, where there are many potential areas of conflict between your application and Progress database client code (including signal handling, global name, and memory space conflicts), an ESQL small client virtually eliminates these conflicts because the bulk of Progress code in the executable comes from the client communication modules that you link with your application. These modules do little more than package and send your ESQL requests to the OID, and pass the results from the OID to your application code.

If you cannot place the OID on the same machine as the Progress database server (because of networking throughput, processing performance, or machine accessibility), you can run the OID on the same machine as the small client. You can also run it on a machine entirely separate from both the small client and the database server (see Figure 3–3). However, if the OID is remote from the database server, your network load is at least equivalent to using a remote large client.

The keys to optimal response in a remote small client configuration are:

- A database server machine with sufficient memory and CPU power to efficiently manage the local OIDs (one for each small client) allocated for the application.
- A network that can sustain acceptable data transfer rates for many small client applications running simultaneously.

Figure 3–4 depicts a typical ESQL small client configuration where each ESQL small client communicates with an OID using TCP/IP.

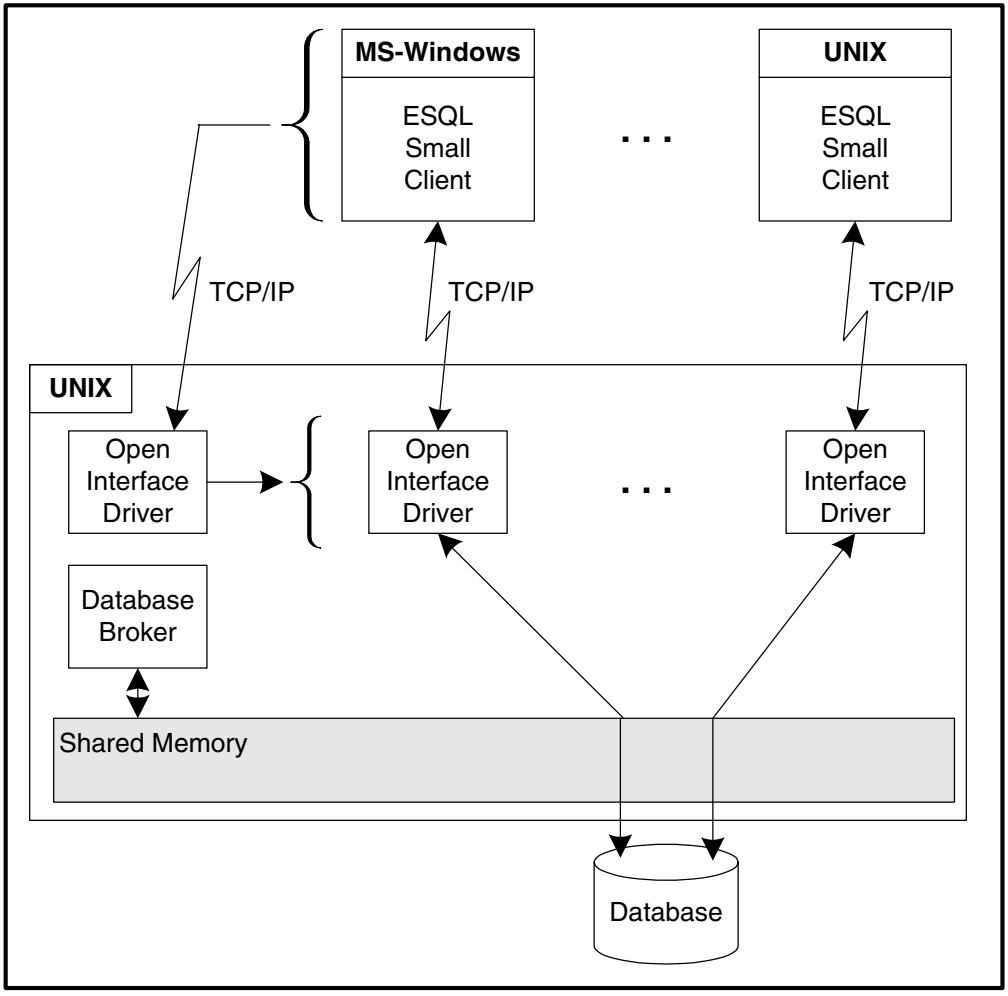

Figure 3–4: Typical Remote Small Client Configuration

Local Configurations

You might also want to run small clients local to OIDs on the database server machine. This configuration is functionally equivalent to the remote small client configuration in Figure 3–4.

3.1.3 Differences Between Small and Large Clients

The OID for small client database access is installed with a fixed set of standard capabilities that include:

- The ability to connect to local Progress databases or remote Progress database servers using any Progress-supported protocol on the OID platform.
- The SQL Compiler to compile run-time SQL requests.
- Progress sort routines.

Unlike large client executables, you cannot customize an OID using the end user configuration (PROBUILD) utility to access a supported DataServer other than Progress, to use your own sort routines, or to access the Host Language Call (HLC) interface from database schema triggers. To include these Progress features in your ESQL application, use a large client executable.

3.2 Overview of Building an ESQL Executable

After you design and program your ESQL application, follow these steps to build an ESQL executable:

1 ♦ If required for your operating system, set up your environment using PROENV, which is provided with your Progress installation.

PROENV sets the environment variables for the sqlcpp preprocessor, C compiler, and linker. For more information on PROENV and whether you need to run it, see the *Progress Client Deployment Guide*.

2 ♦ If your ESQL source files contain Progress/ESQL statements, preprocess your source files with the `sqlald()` preprocessor.

This step produces a preprocessed (pure) C source file. If you run the preprocessor with the precompile option, this step produces an r-code file for each unique static ESQL statement in your application. For more information, see the "Preprocessing Progress/ESQL Source Files" section.

NOTE: You do not have to preprocess ESQL source files that call ESQL-LIB directly and contain no Progress/ESQL statements.

3 ♦ Compile your pure C source files.

This step produces an object file for each source file. For more information, see the "Compiling ESQL Source Files" section.

4 ♦ Link your ESQL executable.

Use the link script that the PROBUILD utility generates for all ESQL clients except the Windows small client.

NOTE: To link the Windows small client, use your own linker response file.

If you are using Microsoft Visual Studio® to build your ESQL application, add `sqlald()` (located in `%DLC%\probuild\esqlc`) to the `objs/line`. The companion file, esql32.dll, is located in `%DLC%\bin`.

The PROBUILD utility automatically generates a UNIX link script or Windows linker response file based on:

- The ESQL client or preprocessor product that you want to build.
- The Progress configurable elements to include in your executable, such as communications protocols or user-defined sort routines and DataServers to non-Progress databases (large client and preprocessor products only).
- The object modules to include for your ESQL application.

For more information on linking all ESQL client executables and PROBUILD, see the "Linking ESQL Executables" section.

3.3 Preprocessing Progress/ESQL Source Files

The SQL preprocessor (`sqlcpp`, located in your ESQL installation directory) does the following:

- accepts as input a Progress/ESQL source file that contains a mixture of Progress/ESQL and C language statements
- converts the ESQL statements into calls to ESQL-LIB functions
- produces a pure C source file with optional r-code for any static ESQL statements

Unless you specify other paths, the preprocessor places its output into your current working directory.

You can run the preprocessor in one of two modes: precompile mode and non-precompile mode.

NOTE: You do not have to preprocess ESQL source files that contain no Progress/ESQL statements and call ESQL-LIB directly.

3.3.1 Precompile Mode

When you run the preprocessor in precompile mode (using the -p startup parameter), it compiles and generates r-code files for various static ESQL statements. You can precompile only Data Manipulation Language (DML) statements. The preprocessor does not precompile Data Definition Language (DDL) statements. Instead, the preprocessor passes these statements to the ESQL-LIB interface for compilation and execution at application run time. If you run the preprocessor in precompile mode, it displays a warning message if it finds any static DDL statements in the file.

CAUTION: Any DML statements that you precompile before DDL statements actually execute (and change the schema) have the potential for cyclic redundancy check (CRC) conflicts during execution. Therefore, avoid preprocessing in precompile mode if your application modifies the schema.

Database Connection

To precompile your SQL, the preprocessor must connect to the database(s) that you want to access with your ESQL application. Therefore, start all database servers that you need before running the preprocessor.

Supply the preprocessor with the correct startup parameters to connect to these databases, including all database connection parameters that you pass to the sqlconn() ESQL-LIB function in your application. In precompile mode, the preprocessor acts as a Progress database client. For more information on database connection parameters, see the *Progress Startup Command and Parameter Reference*.

NOTE: Be sure that your SQL preprocessor contains all the network protocols and Progress DataServer capabilities that you require to compile and run static ESQL. Use the PROBUILD utility to reconfigure a new preprocessor, if necessary. For more information on PROBUILD, see the *Progress Client Deployment Guide*. If you are precompiling for a Progress DataServer database, connect to the schema-holder database; you do not need to connect to the non-Progress database.

R-code Generation

After connecting to databases, the preprocessor opens the specified ESQL source files, converts the appropriate ESQL statements to ESQL-LIB function calls and data declarations, and precompiles each static ESQL statement it encounters. For some SQL statements, the preprocessor generates an r-code file. (Even if a particular SQL statement appears more than once in the application, the preprocessor only generates a single r-code file for that SQL statement.) The preprocessor also generates code to pass the name of the generated r-code file (rather than the SQL statement itself) to the ESQL-LIB function that invokes the statement.

NOTE: In Version 9, Progress r-code is portable between operating systems and hardware platforms. If you are converting a Version 6 ESQL (HLI/C) application to run with Version 9, preprocess any source files in precompile mode to regenerate any precompiled r-code you want to maintain.

Error Handling

If the SQL Compiler detects an error in a static ESQL statement, it displays an error message on the screen and continues to compile the remaining static ESQL statements in the source file. The preprocessor generates r-code files for as many correct ESQL statements as it can. However, it only creates a preprocessed C source file if no errors are detected.

3.3.2 Non-precompile Mode

When you run the preprocessor in non-precompile mode, it opens the specified ESQL source files and converts the appropriate ESQL statements to ESQL-LIB function calls and data declarations. As it converts ESQL statements to C, the preprocessor generates code to pass the SQL for each static ESQL statement to the ESQL-LIB function that invokes the statement. The ESQL interface then compiles all ESQL statements in your application at run time.

NOTE: If you preprocess in non-precompile mode, build either a full large client executable (a large client that contains the SQL Compiler) or a small client executable (that uses the SQL Compiler built into the OID). A run-time large client cannot execute a non-precompiled ESQL application. For more information, see the "Linking ESQL Executables" section.

3.3.3 ESQL Preprocessor Syntax and Operation

This is the syntax for the SQL preprocessor command. Many of the startup parameters are abbreviated, however, the long form is listed for each parameter that has one:

SYNTAX

```
sqlcpp { input-file [ output-file ] | -fl list-file }
     [ -debug ]
     [ -qf ] [ -esqlnopad ]
     [ [ -a app-name ] -p parameters ]
```

input-file

The filename of the ESQL source code file you want to preprocess (usually includes a `.cc` extension).

output-file

The filename (including any extension) you want to assign to the preprocessed pure C version of your source file. If you omit *output-file,* the preprocessor automatically assigns the name using the filename portion of *input-file* appended to a `.c` extension.

-fl list-file
-flist list-file (long form)

Allows you to preprocess more than one ESQL source file at a time. The *list-file* parameter is the name of a text file that contains a list of filename pairs, one pair per line. Each pair is separated by a space. The first name is the name of the ESQL source file; the second (optional) is the name of the C output file. Omitting the second name causes the same effect as specifying only *input-file* without *output-file* on the command line.

-debug

Causes the preprocessor to provide the text of any static ESQL statements as readable comments. The preprocessor reformats ESQL requests as difficult-to-read character arrays to pass them to ESQL-LIB functions. Using this parameter generates a readable version of the SQL statement for debugging, and places it just before the corresponding function call. The preprocessor also blank-pads math operators (unless you also specify *-esqlnopad*) and translates scientific notation to the equivalent Progress decimal format.

-qf
-qflagger (long form)

Displays warning messages whenever the preprocessor encounters nonstandard SQL in DML or DDL statements. The *-qf* parameter flags nonstandard DML only in precompile mode. It flags nonstandard DDL only in non-precompile mode (the only valid preprocessing mode for DDL statements).

-esqlnopad

Allows you to specify hyphenated names and language elements without escape characters (\ ), as in the Progress 4GL. Unlike Progress fields, the hyphen is not a valid character for SQL column names because SQL always treats a hyphen as a minus sign. The expression qty-used means "qty minus used" in SQL, whereas Progress evaluates it as the field name, "qty-used". To support standard SQL, the SQL preprocessor assumes the following:

- A hyphen is a minus sign; if you type **qty-used** (without the escape character) then qty-used is blank padded to qty-used.
- If you type **qty\-used**, you are referring to the Progress field (column), qty-used.

The *-esqlnopad* parameter causes the preprocessor to treat all hyphens as hyphens and requires the application to ensure that minus symbols are blank-padded. This makes constructing queries against databases with hyphenated column names easier.

-a *app-name*
-applnam *app-name* (long form)

(Precompile mode only.) Allows you to specify an application name that generates the names for your r-code files. The preprocessor uses the first two characters of *app-name* as a prefix for each r-code filename. It assigns the next six characters by generating a hexadecimal number from the text of the compiled SQL request, and adds the .r extension. If you do not specify this parameter, the preprocessor uses only the six hexadecimal characters plus the .r extension to generate the names for r-code files.

The full value of *app-name* must be identical to the application name you use when you call sqllogin() from within your application.

This parameter has no effect in non-precompile mode.

p parameters
-progress parameters (long form)

This parameter performs two functions:

- It indicates that all the parameters that follow are Progress startup parameters.
- It tells the preprocessor to run in precompile mode.

The parameters string must contain all the database connection parameters that your application requires for execution. Be sure that all logical database names are the same as those your application uses when it connects to the databases.

The parameters string can also specify a parameter file using the Parameter File (-pf) startup parameter.

If your application connects to its databases by calling sqlconn(), enter the same database connection parameters on the command line that you enter for the sqlconn() calls in your program. For more information on database connection parameters and parameter files, see the *Progress Startup Command and Parameter Reference*.

3.4 Compiling ESQL Source Files

When you compile ESQL source files for your application, make sure you are using the correct compiler:

- On UNIX, use the `cc` command that comes with your system.
- On Windows, use the `cl` command from Microsoft Visual C++ V6.0 or later.

3.4.1 Compiling on UNIX Systems

Following is a typical compiler command line for C source files on UNIX systems:

```
cc -c -I$DLC/probuild/esqlc clidemo.c
```

This is how the command uses the command line parameters:

- The `cc` command performs a variety of functions including linking an executable. The `-c` parameter suppresses the link process so that the command generates only an object file.
- The `-I` parameter informs the compiler where it can find the `proesql.h` header file needed to define ESQL declarations in your application. The example shows the default

installation directory for ESQL where this header file is located. If `proesql.h` resides in a different directory on your system, specify that directory instead.

3.4.2 Compiling in Windows

Following is a typical compiler command line for C source files in Windows:

```
cl /c /I%DLC%\probuild\esqlc clidemo.c
```

This is how the command uses the command line parameters:

- The `cl` command performs a variety of functions including linking an executable. The `/c` parameter suppresses the link process so that the command only generates an object file.
- The `/I` parameter informs the compiler where it can find the `proesql.h` header file needed to define ESQL declarations in your application. The example shows the default installation directory for ESQL in which this header file is located. If `proesql.h` resides in a different directory on your system, specify it instead.

Using proesql.h Function Prototypes

If you are using the ESQL function prototypes defined in the `proesql.h` header file, define either the _WINDOWS or FPROCS C preprocessor constants to the compiler. You can define these constants with the /D option (for example, /DFPROCS) or specify the /GA option to automatically define the _WINDOWS constant.

3.5 Building Applications Containing Windows Resources

Use the resource compiler to integrate your application executable with Windows resources to create a Windows executable for your application.

The resource compiler performs the following tasks:

- It compiles the resource-definition file and resource files (such as icon and cursor files) that you prepare for your application into a binary resource (`.res`) file.
- It combines your initial application executable with the binary resource file to produce the Windows executable for your application.

For example, the resource file for `wesqldmo` can be complied with the following command:

```
rc wcesql.rc
```

This produces a binary resource file (`.res`) that must be included in your link script. For a Large Client, the `.res` file can be added in the object files step in PROBUILD, along with the object file names. For a Small Client, the `.res` file can be added to the link script along with the object file names.

For more information on using the Microsoft Linker and Windows Resource Compiler, see your Microsoft software tools documentation.

3.6 Linking ESQL Executables

If you are preparing an existing ESQL application written before Progress Version 7, you might only have to preprocess, compile, and link your application using the appropriate link script. To modify your application for Version 7, Version 8, or Version 9, preprocess and compile the necessary modules.

3.6.1 ESQL Client Executables

You can link your application as one of two different types of ESQL client executables, depending on your operating system, network configuration, and application requirements. Following are the two types of ESQL client executables:

- **ESQL Large Client (Full)** — This is a complete Progress database client executable including the SQL Compiler, single or multi-user capability, and the ability to run locally or remotely to a database server. This executable can handle all static and dynamic SQL requests and include Progress options such as DataServers, user sort routines, and calls to HLC in database schema triggers.
- **ESQL Small Client** — This is the preferred ESQL client for a multi-user network operation. It contains only Progress client communications and minor utility modules to access the Progress Open Interface Driver. The OID can handle all static and dynamic SQL requests on behalf of an ESQL small client.

You can use the PROBUILD utility to generate UNIX link scripts or Windows linker response files for all ESQL client executables except the Windows small client. For the Windows small client, you create your own link script.

3.6.2 Linking for Windows with ESQL Small Client

ESQL provides a Dynamic Link Library (DLL) called `esql32.dll` (`esql32.lib`) for Windows applications that use the ESQL Small Client. This contains the Progress client communications and utilities functions to access the Progress OID.

The ESQL Small Client DLL (`esql32.dll`) contains client communications modules that include Winsock-compatible TCP/IP.

Preparing the Linker Response File

To link your Windows application with the ESQL Small Client, construct a standard Windows linker response file for your application, and include a reference to the import library (`esql32.lib`) for the ESQL small client DLL.

Figure 3–5 shows a sample linker response file (`smallcl.lnk`) for an ESQL small client application (`inventory.*`). The file references include the necessary path prefixes to locate your installation files.

```
inventory.obj

%DLC%/probuild/esqlc/esql32.lib
/NOLOGO
/OUT:inventory.exe
/MAP:inventory.map
libcmt.lib
kernel32.lib
shell32.lib
user32.lib
comdlg32.lib
gdi32.lib
advapi32.lib
comctl32.lib
winspool.lib
netapi32.lib
wsock32.lib
```

Figure 3–5: Sample Linker Response File

Building Your Small Client Application for Windows

After you create your linker response file, complete building your Windows Small Client application. For example, to complete building the small application `inventory.*` using the sample linker response file `smallcl.lnk`, enter this command:

```
Link @smallcl.lnk
```

This creates the executable (`.EXE`) file for your application.

3.6.3 PROBUILD and the Linker

For all ESQL executables other than Windows small clients, use the PROBUILD utility to construct a link script appropriate for your operating system and configuration requirements. As part of the script construction procedure, PROBUILD lets you select:

- The ESQL client executable to build
- Any user configurable elements to add, including DataServers
- User sort routines
- HLC modules for database schema triggers

Just before generating the script, PROBUILD prompts for and includes the names of your ESQL object modules, and inserts them in order to properly link with the required Progress object modules for your selected executable.

PROBUILD constructs link scripts for many types of Progress executables other than ESQL clients. However, the general procedure is the same for all of them. You can also add application object file references directly in the link script using a text editor.

ESQL large clients can run as either Console or Windows applications. PROBUILD will automatically generate an ESQL large client link script for console applications by default on Win32. If your application uses Windows programming, you need to edit the generated link script manually. To do this, change the lines in `wcesql.lnk` as follows:

From:

```
/SUBSYSTEM:console, 4.0
/ENTRY:mainCRTStarup
```

To:

```
/SUBSYSTEM:windows,4.0
/ENTRY:WinMainCRTStartup
```

After you construct a link script, build your small or large client executable using your system linker. For information on using PROBUILD and your system linker to build your ESQL executable, see the *Progress Client Deployment Guide*.

3.7 Running ESQL Applications

The procedure for running an ESQL application as a small or large client is different, although they provide the same SQL functionality. You can execute an ESQL large client like any Progress database client, using many of the same startup parameters to manage database access. Many of these parameters (especially those used for database connection) also apply to starting small clients.

For more information on running Progress database clients, see the *Progress Client Deployment Guide*. For information on running ESQL small clients, see the "Starting ESQL Small Clients" section. For information on startup considerations common to both small and large clients, see the "Preparing to Run Small and Large Client Applications" section.

3.7.1 Preparing to Run Small and Large Client Applications

Before running an ESQL small or large client application, make the following preparations:

- Compile the 4GL (generate the r-code) for any database schema trigger procedures required by your application.
- Ensure that the PROPATH environment variable is properly set to include all r-code required by your application.
- Choose the appropriate ESQL startup parameters for your application:
  - If you want all ESQL messages written to your database log file as well as the message queue, use the ESQL Log (-esqllog) startup parameter with your application.
  - If you want to use hyphenated field (column) names and language elements in non-precompiled or dynamic ESQL requests just as they are interpreted by the Progress 4GL, use the ESQL No Padding (-esqlnopad) startup parameter with your application.
  - If your application must meet ANSI SQL standards, use the startup parameter (ANSI SQL (-Q) or ANSI SQL Client (-Q2)) that sets your required level of ANSI SQL compatibility.

For information on compiling 4GL procedures, such as those used for database schema triggers, see the information on the Application Compiler in the *Progress Basic Development Tools* manual.

The PROPATH for a small client OID or large client executable must contain the pathname(s) where you store any r-code for your application, including compiled schema trigger procedures and precompiled SQL requests. Even if your application includes the SQL Compiler, it cannot

recompile and execute SQL requests that have already been precompiled because the text of SQL requests is no longer available after the r-code has been precompiled.

Table 3–1 describes the ESQL startup parameters that provide common execution options for both small and large clients.

Table 3–1: Common ESQL Startup Parameters *(1 of 2)*

Startup Parameter	Description
ESQL Log (-esqllog)	Sends Progress messages generated for ESQL applications to the database log (`.lg`) file as well as the message queue for the ESQL-LIB function `sqlgetmsg()`.
ESQL No Padding (-esqlnopad)	Allows you to specify hyphenated names and language elements without the escape character (\), as in the Progress 4GL. The hyphen is not a valid character for SQL column names because SQL always treats a hyphen as a minus sign. The expression qty-used means "qty minus used" to SQL, whereas Progress evaluates it as a field name, qty-used. To support standard SQL, the SQL scanner assumes the following: • A hyphen is a minus sign; if you type **qty-used** (without the escape character), then qty-used gets blank-padded to qty-used. • If you type **qty\-used**, you are referring to the Progress field (column) name, qty-used. The -esqlnopad parameter causes the SQL scanner to treat all hyphens as hyphens (as in the Progress 4GL) and requires the application to ensure that minus symbols are blank-padded. This makes constructing queries against databases with hyphenated column names easier.

Table 3–1: Common ESQL Startup Parameters *(2 of 2)*

Startup Parameter	Description
ANSI SQL (-Q)	Provides support for ANSI SQL89 Level II requirements, including: • All character fields return to your application blank-padded when the output string is shorter than the storage allocated for it. • The maximum number of digits to the right of the decimal point increases from 10 to 15. • Division by zero in arithmetic expressions returns an error rather than the UNKNOWN value. The performance of decimal field manipulation degrades significantly when running with this parameter.
ANSI SQL Client (-Q2)	Sets the environment to conform to rules governing the ANSI SQL89 standard with its guaranteed ability to serialize transactions. This feature is an addition to the -Q startup parameter. The -Q2 addition allows one client exclusive access to the database. Other connections pend until the active connection completes. Use this ANSI feature only if you need to guarantee serialized SQL transactions in ESQL. If you want concurrent database access, exclude -Q2 from the SQLLOGIN statement.

3.7.2 Starting ESQL Small Clients

Starting an ESQL small client application requires a priority of events similar to starting any Progress multi-user application. Following is the typical sequence of steps.

1 ♦ Set the variables with PROOIBRK and PROOIDRV:

On UNIX:

```
PROOIBRK=$DLC/bin/_prooibk
export PROOIBRK
PROOIDRV=$DLC/bin/_prooidv
export PROOIDRV
```

On Windows:

```
set PROOIBRK=%DLC%/bin/oibrkr32.exe
set PROOIDRV=%DLC%/bin/oidrvr32.exe
```

2 ♦ For a multi-user application, start any required database brokers or servers. For example:

```
proserve demo
```

3 ♦ If the OIB/OID is remote from the ESQL small client, start the Open Interface Broker. The OIB then starts an Open Interface Driver for each remote small client. For example:

On UNIX:

```
$PROOIBRK -SV -S demosv -H localhost -N TCP
```

On Windows:

```
%PROOIBRK% -SV -S demosv -H localhost -N TCP
```

Depending on your run-time configuration, some of these steps may change or be machine dependent.

For more information on Step 2, see the *Progress Startup Command and Parameter Reference*. The following sections describe how to complete Step 3.

Open Interface Driver on UNIX Systems

If you plan to run the OID on a remote UNIX or other multi-process machine, start the OIB on that machine to listen for remote connections from small clients. This is true whether or not the ESQL small client runs on Windows or UNIX systems.

3.7.3 Connecting Small Clients to Open Interface Drivers

To identify the connection between each ESQL small client and its OIB/OID, Progress provides an OID Connection (-SV) startup parameter. This parameter introduces a list of additional parameters that specify the connection between the small client and the OIB/OID. This connection parameter list directly follows the -SV parameter on an OIB or ESQL small client command line up to the next -db parameter or end-of-line character. Only the small client can specify database connection parameters, and any -db parameter list(s) it specifies can appear before or after the -SV parameter list on the command line. The OIB passes those parameters that affect each OID (including database connection parameters) to the OID.

You can only include one -SV parameter list on each OIB and small client command line. If a Parameter File (-pf) startup parameter is included in the -SV list, all parameters in the file, up to the first -db, are part of the -SV parameter list. However, all parameters after and including the first -db in the parameter file are ignored, and are not passed to OIDs. The only parameters you can effectively put in this parameter file are for the OID connection.

Table 3–2 lists the startup parameters that you can include in a -SV parameter list for both small clients and the OIB. See the *Progress Startup Command and Parameter Reference* for a complete description of each parameter.

Table 3–2: OID Connection (-SV) Parameter List *(1 of 2)*

Startup Parameter	Description
-H	Host Name of the OIB
-m1[1]	Auto Server (passed to OIDs by the OIB)
-m2[2]	Manual Server (specified for a user-started OID server only)
-Mm	Message Buffer Size
-N	Network Type
-pf	Parameter File

Table 3–2: OID Connection (-SV) Parameter List *(2 of 2)*

Startup Parameter	Description
-pp	PROPATH used to modify the PROPATH of an OID so that it may locate existing r-code files.
-S	Service Name for the OIB

[1] No need for you to specify—the OIB automatically provides it.

[2] Required if you start an OID directly. For more information, see the "Starting UNIX Open Interface Brokers" section.

OID Startup and the Progress Client Environment

In addition to the OID connection parameters, specify all Progress session-wide client parameters when you start the OIB (or an OID directly), such as the Number of Databases (-h), No Lock (-NL), and Record Buffer Size (-Mr) startup parameters. The ESQL small client cannot pass these parameters to the OID once it has started. Essentially, each OIB establishes a common Progress client environment for all OIDs that it spawns. The one exception to this is when starting a local small client that auto-starts the OID. For more information, see the "Starting ESQL Small Clients" section.

The following sections describe how to start an OIB and ESQL small client using -SV and -db parameter lists.

3.7.4 Setting OIB and OID UNIX Environment Variables

Before you run a small client application, you may have to set the environment variables listed in Table 3–3.

Table 3–3: OIB and OID UNIX Environment Variables

Environment Variable	Setting
PROOIBRK	The pathname of the OIB executable, `$DLC/bin/_prooibk` by default.
PROOIDRV	The pathname of the OID executable, `$DLC/bin/_prooidv` by default.

When you set these variables yourself, set them on the machine where you run the OIB and OID. For example, to set PROOIDRV in the UNIX C shell (csh), you can enter this command at the prompt:

```
setenv PROOIDRV $DLC/bin/_prooidv
```

To set PROOIDRV in the UNIX Bourne shell (sh), you can enter this command at the prompt:

```
PROOIDRV=$DLC/bin/_prooidv; export PROOIDRV
```

3.7.5 Starting UNIX Open Interface Brokers

If your OIB is on a remote UNIX system, you can start the OIB by executing the prooibrk shell script provided with your Version 9 installation. The OIB then waits for login requests from small clients, and starts an OID for each one. If you run the OIB on the database server machine (the recommended configuration), all OIDs are started as self-service clients of the local database(s) they access. (If the OIB is remote from the database server, the small client must specify the network connection to the remote server machine as part of the database startup parameter list for that database. For more information, see the "Starting ESQL Small Clients" section.)

As a minimum, use this syntax to start an OIB:

SYNTAX

```
prooibrk -SV -S service-name -H host-name -N TCP
```

service-name

The connection name to the OIB that small clients use. It is also a name in the TCP/IP services file of the OIB machine, or is registered with your Network Information Service (NIS).

3.7.6 Starting an OID Directly Without a Broker

You can start a remote OID directly without using a broker. You might want to do this for monitoring purposes, to identify the exact connection between an ESQL small client and its OID. You start an OID with the addition of the Manual Server (-m2) startup parameter, as shown in the following examples:

On UNIX:

```
$PROOIDRV -SV -S esqlsv1 -H host-name -N TCP -m2
```

On Windows, there is no OIB. You can start the OID directly with the following command:

```
%PROOIDRV% -SV -S esqlsv1 -N TCP -H host-name -m2
```

3.7.7 Starting the Small Client Processes

When you start an ESQL small client, ensure that any precompiled r-code files are in the PROPATH of the OID that runs them.

Specifying the Small Client Command

As a minimum, use the startup parameters on the following command line to start a remote ESQL small client with precompiled r-code using TCP/IP:

SYNTAX

```
client-name -SV -S service-name -H host-name
```

client-name

Your ESQL application filename.

service-name and *host-name*

The name of the OIB (or manually started OID) started for the small client and of the remote host. You define both names in the TCP/IP services and hosts files of the OIB/OID machine.

If your application has precompiled r-code files that are not in the PROPATH of the OID, use the -pp parameter to identify the pathname of the files. You can also specify other common ESQL, OID connection, and database connection parameters that your application requires.

Specifying Databases at Startup

The ESQL small client also specifies the database(s) to which its OID connects and the connection parameters. To specify databases from the command line, you can start a small client as follows:

```
inventory -db db1 -SV -S sqlsv1 -H db1machine
```

In this example, a small client, `inventory`, accesses the `db1.db` database through an OID that the OIB, `sqlsv1`, starts. The OID runs on the database server machine, `db1machine`, accessing the database directly:

```
inventory -db db1 -S db1sv -H db1machine -SV -S sqlsv1 -H sqlmachine
```

In this example, the OID runs on `sqlmachine` accessing the database remotely through the database broker/server, `db1sv`, on `db1machine`. The small client is started on a third machine, which is why it specifies a host name for both the OIB/OID and the database broker/server. Although not a recommended configuration, this example shows how to start a small client in a three-tier, remote configuration with the small client, OID, and database broker/server running on separate machines in a network.

3.7.8 Starting Local Small Clients

If both the ESQL small client and OID are on the same machine, there is no need to start the OIB or OID prior to starting the small client. The OID is started automatically (auto-started) for the ESQL small client. This is the basic syntax for starting local small clients:

SYNTAX

```
client-name -SV -S service-name -N TCP -H host-name
```

client-name

Your ESQL application filename.

service-name

The name of the auto-started OID.

For each operating system, the server name (-S) for your OID is required.

NOTE: Unlike a manually-started or OIB-started OID, when you auto-start an OID with a local small client, provide all session-wide client parameters when you start the small client.

3.7.9 Organizing OID and Database Startup Parameters

One way to organize OID Connection (-SV) and Database Connection (-db) parameters is to put them in different parameter files and specify them as shown in the following example:

```
inventory -SV -pf esqlprms.pf -db db1 -pf dbprms.pf
```

In this example, all connection parameters to the OID for `inventory` are located in the `esqlprms.pf` parameter file, and all database connection parameters are located in the `dbprms.pf` parameter file.

Alternatively, you can place your startup parameters in the `startup.pf` file located in your Progress installation directory (the DLC directory) or create a new `startup.pf` file in another directory and set the PROSTARTUP variable to that directory.

3.7.10 Shutting Down an ESQL Small Client Application

Typically, to shut down an ESQL small client application, the client calls the sqllogout() function, which sends a message to the client's OID to shut itself down. After it is clear that small clients are no longer logging in through an OIB, you can shut the broker down using the PROSHUT utility with the following syntax:

SYNTAX

```
proshut -SV -S service-name [ -H host-name ] [ -N TCP ]
```

service-name

The name of the OIB that the -S parameter provides during broker startup.

Shutting Down Detached Open Interface Drivers

Occasionally, a small client terminates abnormally, leaving its OID running. If you know the service name of the OID (you started it manually):

1 ♦ Locate the appropriate command (**prooidrv -SV -S ...**).

2 ♦ PID with the UNIX **ps** command.

3 ♦ Shut down the OID with the **kill** command.

Otherwise, you must wait until all small clients terminate. Then use **ps** to locate any remaining OID commands (**prooidrv ...**) and shut them down using the **kill** command.

3.8 The ESQL Demo Applications

The next two sections describe how to build and run the static ESQL and the dynamic ESQL demo applications. In addition, the application `clidemo.c` provides a rough demonstration of dynamic parameter markers using direct calls to ESQL-LIB. You only need to compile and link this application.

The ESQL source files for these applications are in your ESQL installation directory.

3.8.1 Static Embedded SQL Demo Applications

The `esqldemo.cc`, `esqldem64.cc` or `wesqldmo.cc` (Windows only) application demonstrates using various static ESQL statements. The `esqldemo.cc` application accesses the sports database, selects several rows from the customer table, and uses a cursor to manipulate a single row at a time. For most of the rows selected, the max-credit field increases by $100 and the row is updated. The last two rows are deleted.

The output of the static ESQL demo application displays information about the rows that have been updated and a message when a row has been deleted. The program also displays a message to indicate when a transaction is being rolled back. None of the updates and deletions are applied, since you might want to run this demonstration again.

The `wesqldmo.cc` application accesses any Progress database and illustrates how you can drive ESQL requests using Windows widgets. It reads the Progress metaschema of the chosen database to produce lists of files and lists of fields within files, as you direct.

Follow these steps to run either static ESQL demo application.

1 ♦ Copy all the files in the ESQL directory to a working directory. The default ESQL directory is `$DLC/probuild/esqlc` for UNIX and `%DLC%\probuild\esql` for Windows.

2 ♦ Change your current directory to the working directory.

3 ♦ Create a copy of the Progress sports database in your working directory with the PRODB utility. For more information on the PRODB utility, see the *Progress Database Administration Guide and Reference*. If you plan to run the `esqldemo.cc` application as a small client, place a copy of the sports database procedure library (`sports.pl`) in the PROPATH of your Open Interface Driver.

4 ♦ Build an ESQL executable from the ESQL source file (`esqldemo.cc`, `esqldem64.cc`, or `wesqldmo.cc`) by preprocessing, compiling, and linking as described in this chapter.

If you preprocess in precompile mode, include -a esqldemo, -a esqldem64 or -a wesqldmo as an SQL preprocessor parameter to set the application name correctly (the call to `sqllogin()` in the demo application).

When you run PROBUILD, set the name of your executable image to `esqldem64` or `wesqldmo.exe` for the command in Step 5 to work. For `wesqldmo`, you will also need to include both `wesqldmo.obj` and `wcesql.res` in the link script. See the "Building Applications Containing Windows Resources" section.

5 ♦ Run `esqldemo` and `wesqldmo` (Windows only) with commands like these. Add appropriate -SV parameters if you are running in the small clients model. The `esqldmo` command prompts for the database to connect:

```
esqldemo -db sports -1 -trig sports.pl
wesqldmo
```

3.8.2 Dynamic Embedded SQL Demo Applications

The `dyndemo.cc` and `clidemo.c` applications demonstrate using various dynamic ESQL statements. When run, the application prompts you to enter an SQL request. Enter your request all on one line and press **RETURN**. No statement terminator is needed; the **RETURN** ends of the request. The request is processed and the results are displayed. You are then prompted for another request. To end the demonstration, type **QUIT**.

Follow these steps to run the dynamic ESQL demo application.

1 ♦ Copy all the files in the ESQL directory to a working directory. The default ESQL directory is `$DLC/probuild/esqlc` for UNIX and `%DLC%\probuild\esql` for Windows.

2 ♦ Change your current directory to the working directory.

3 ♦ Create a copy of the Progress sports database in your working directory with the PRODB utility. For more information on the PRODB utility, see the *Progress Database Administration Guide and Reference*. If you plan to run the application as a small client, place a copy of the sports database procedure library (`sports.pl`) in the PROPATH of your Open Interface Driver.

4 ♦ Build an ESQL executable from the dynamic ESQL source file (`dyndemo.cc` or `clid-emo.c`) by:

- Preprocessing (`dyndemo.cc` only). The `clidemo.c` application requires no preprocessing. It calls directly to the esql32 library functions.
- Compiling
- Linking, as described in this chapter

NOTE: If you plan to run a large client, when you run PROBUILD, select ESQL LARGE CLIENT (FULL) from the product list to build an executable that can compile dynamic ESQL requests. Also, whichever executable you build, set the name of your executable image to `dyndemo` for the command in Step 5 to work.

5 ♦ Run the `dyndemo` executable against the sports database. For example, on UNIX, run `dyndemo` with the following command. Add the appropriate -SV parameters if you are running small clients:

```
dyndemo -db sports -1 -trig sports.pl
```

4

ESQL-LIB Reference

This chapter provides information on the ESQL-LIB interface, its C function calls, and requirements for using them.

Although the function descriptions in this chapter contain usage examples, you can find more complete examples in the sample ESQL applications (`esqldemo.cc`, `esqldem64.cc` or `wesqldmo.cc`, `dyndemo.cc`, and `clidemo.c`) located in your installation directory. The header file `proesql.h` contains the required ESQL data and function declarations for using the ESQL-LIB interface. For more information on ESQL data structures and the sample applications, see Chapter 2, "Programming with ESQL."

This chapter contains:

- An overview of ESQL-LIB, including data and function components.
- A reference entry for each ESQL-LIB function.

NOTE: The reference entries for ESQL-LIB session management and Dynamic ESQL functions provide sufficient information to use these functions directly in your ESQL application. The information on Static ESQL functions describes how the SQL Preprocessor uses them to support Static ESQL statements.

4.1 Overview

The ESQL-LIB interface provides the functions required to execute SQL statements from an ESQL application. This includes functions that you must call directly, as well as functions that the SQL Preprocessor calls to support the ESQL statements that you code (EXEC SQL...;).

The reference entries for these functions describe variables and data structures that your ESQL module must declare. The following sections briefly describe the data objects required to call ESQL-LIB functions. When you use the SQL Preprocessor, it declares and initializes most of the data objects that ESQL-LIB requires when it converts your Progress/ESQL statements into function calls. If you make direct calls to ESQL-LIB, you must manage all required variables, data structures, and literal values.

4.1.1 Error and Status Returns

Most functions of the ESQL-LIB interface return specific exception conditions. These conditions are returned to a variable that you declare called SQLCODE or SQLSTATE. If you use the Preprocessor, you must declare SQLCODE or SQLSTATE:

```
long SQLCODE;
char SQLSTATE[6];
```

The Preprocessor references SQLCODE or SQLSTATE for each ESQL statement that can return an ESQL exception. If you use ESQL-LIB directly, you must explicitly bind SQLCODE or SQLSTATE to ESQL prior to each function that sets a value for SQLCODE or SQLSTATE using the `sqlcdbind()` function. For more information on using SQLCODE or SQLSTATE and their possible values, see Chapter 2, “Programming with ESQL.”

4.1.2 SQL Request Handles

Many ESQL-LIB functions take an SQL request handle as a parameter. This request handle references either an open cursor in Static ESQL or a prepared statement in Dynamic ESQL. In Dynamic ESQL, a request handle is also the ESQL-LIB equivalent of a Progress/ESQL statement identifier. ESQL-LIB uses this handle to reference the same SQL object for many calls to the interface. The Preprocessor declares a unique request handle for each Static ESQL cursor and each Dynamic ESQL prepared statement using the following specification:

```
static SQLRQHDL *sqln = (SQLRQHDL *) 0;
```

The *n* component of the name is an ascending integer starting at zero. SQLRQHDL is defined in `proesql.h`, the header file located in your ESQL installation directory. If you use ESQL-LIB directly, you must declare your own request handle variables. The contents of a request handle is undefined after a close statement. Although `proesql.h` defines SQLRQHDL as a pointer to char, the request handle actually references an ESQL internal structure that maintains the state of the prepared statement. It has no meaning to your application.

4.1.3 Static ESQL Variables and Literal Constants

For Static ESQL, the Preprocessor declares several variables and literal constants that specify global information or are passed as parameters to Static ESQL-LIB functions. Table 4–1 lists these data components and their usage as the Preprocessor defines them.

Table 4–1: Static ESQL Variable Declarations

Variable or Literal Constant	ESQL-LIB Reference	Usage
static char *sqlv[]	psqlv	Specifies the host variables to which the current SQL request refers.
static char sqltxt[]	prequest	Passes the text of the specified SQL request to a Static ESQL-LIB function. If the request is precompiled, this is a literal character string specifying the name of the r-code file that is compiled for the request.
static char sqlfmt[]	pvformat	Specifies the format of each host variable that `psqlv` specifies to a Static ESQL-LIB function.
Literal character strings	pcursnam prcfile	Passes the specified cursor name. Passes the name of the precompiled r-code file that implements some specific ESQL-LIB functions (or a null string if the request is not precompiled).
Literal integers	flag	Passes ESQL-LIB execution options for a Static ESQL-LIB function.

4.1.4 Dynamic ESQL and the SQL Descriptor Area

Many Dynamic ESQL statements require a data structure (SQL descriptor area) to describe and exchange data between your application and the ESQL-LIB interface. This data structure takes the place of C variables declared for the same purpose in Static ESQL statements.

The C structure definition for this SQL descriptor area is contained in the `proesql.h` header file in your ESQL installation directory. It consists of a single SQLDA_T structure that includes an array of pointers to SQLVAR_T structures, one for each table column or parameter marker. You must declare a pointer to each SQL descriptor area to use. An SQL descriptor area is not bound to a particular SQL request, but is the mechanism by which data is exchanged. You can use a single SQL descriptor area to fetch values from one table and insert those values in another. For more information on the SQL descriptor area, see Chapter 2, "Programming with ESQL."

4.1.5 ESQL-LIB Functions

Table 4–2 describes each of the functions in the ESQL-LIB interface. The middle column indicates whether a function can affect the value of the SQLCODE or SQLSTATE status variable.

Table 4–2: ESQL-LIB Functions *(1 of 4)*

Function	SQLCODE and SQLSTATE	Description
`sqlald()`	√	Allocates storage for an SQL descriptor area used to implement Dynamic cursor statements. Supports the EXEC SQL ALLOCATE DESCRIPTOR... statement.
`sqlalloc()`	–	Allocates data storage for columns in an SQL descriptor area.
`sqlapplname()`	√	Sets the ESQL application name.
`sqlcdbind()`	–	Binds the address of your SQLCODE or SQLSTATE variables to ESQL in support of error and status reporting on each ESQL statement execution.
`sqlcommit()`	√	Commits any pending database transaction. Supports the EXEC SQL COMMIT WORK statement.
`sqlconn()`	√	Connects to one or more databases using Progress database connection parameter.
`sqldelrow()`	√	Deletes the row at the current cursor position. Supports the Static positioned DELETE statement.

Table 4–2: ESQL-LIB Functions *(2 of 4)*

Function	SQLCODE and SQLSTATE	Description
`sqldinit()`	–	Initializes an SQL descriptor area.
`sqldiscon()`	√	Disconnects a database.
`sqldld()`	–	Frees the storage previously allocated with a call to `sqlald()`. Supports the EXEC SQL DEALLOCATE DESCRIPTOR ... statement.
`sqldsize()`	–	Calculates the memory size required to allocate an SQL descriptor area for a specified number of columns.
`sqldynclose()`	√	Closes a Static cursor specification (cursor SELECT statement), and resets (for repeated execution) or closes a Dynamic ESQL statement or cursor specification. Supports the Static or Dynamic EXEC SQL CLOSE... statement.
`sqldyndesc()`	√	Obtains column description information for a Dynamic ESQL result set. Supports the Dynamic EXEC SQL DESCRIBE... statement.
`sqldyndescinp()`	√	Obtains description information for all parameter markers in a Dynamic ESQL statement.
`sqldynexec()`	√	Executes a prepared Dynamic ESQL statement. Supports both the Dynamic EXEC SQL OPEN... and EXEC SQL EXECUTE... statements.
`sqldynftch()`	√	Fetches the next row of a Dynamic cursor. Supports the Dynamic EXEC SQL FETCH... statement.
`sqldyngetn()`	√	Retrieves the cursor name currently associated with a prepared Dynamic ESQL statement.

Table 4–2: ESQL-LIB Functions *(3 of 4)*

Function	SQLCODE and SQLSTATE	Description
`sqldynncols()`	√	Retrieves the number of columns in a Dynamic ESQL result set.
`sqldynnparms()`	√	Retrieves the number of parameter markers in a prepared Dynamic ESQL statement.
`sqldynposfetch()`	√	Fetches the specified row of a dynamic open cursor. Returns normalized values.
`sqldynprep()`	√	Prepares a Dynamic ESQL statement. Supports the Dynamic EXEC SQL PREPARE... statement.
`sqldynrq()`	√	Dynamically compiles and executes SQL requests other than cursor specifications. Supports the Dynamic EXEC SQL EXECUTE IMMEDIATE... statement.
`sqldynsetn()`	√	Associates a cursor name with a prepared Dynamic ESQL statement. Supports the Dynamic EXEC SQL DECLARE... statement.
`sqlfetch()`	√	Fetches the next row of a Static ESQL cursor. Supports the Static EXEC SQL FETCH...INTO... statement.
`sqlfree()`	–	Frees the storage previously allocated with a call to `sqlalloc()`.
`sqlgetmsg()`	–	Retrieves the next message in the Progress message queue.
`sqlgetrqhdl()`	√	Retrieves a request handle corresponding to a prepared Dynamic ESQL statement identifier.
`sqllogin()`	–	Starts a Progress ESQL session using Progress startup parameters.
`sqllogout()`	√	Terminates the Progress ESQL session.

Table 4–2: ESQL-LIB Functions *(4 of 4)*

Function	SQLCODE and SQLSTATE	Description
`sqlrequest()`	√	Compiles and executes any Static ESQL statement or cursor specification, except the Static FETCH, positioned DELETE, and positioned UPDATE statements.
`sqlrollback()`	√	Rolls back any pending database transaction. Supports the EXEC SQL ROLLBACK WORK statement.
`sqlsetlogoption()`	√	Sets a session-wide option for the ESQL interface.
`sqlupdrow()`	√	Updates the row at the current cursor position. Supports the Static positioned UPDATE statement.

4.1.6 General Diagnostics

Most normalized functions return the status values described in Table 4–3.

Table 4–3: Normalized Function Returns

Return Value	Description
0	Successful completion.
-1	General failure. Check the SQLCODE or SQLSTATE[1] status variables and call `sqlgetmsg()` for more information
-2	No SQL session active. You must make a prior call to `sqllogin()` to use this function.
-3	An attempt to reenter ESQL-LIB was encountered. ESQL-LIB is not reentrant or thread safe.
100	No data outcome.

[1] The value returned in SQLCODE or SQLSTATE provides a more specific indication of the error. For more information on SQLCODE and SQLSTATE, see Chapter 2, "Programming with ESQL."

The exceptions to these status values include functions that either return other functional data or do not return any values. These functions and their return values are listed in Table 4–4.

Table 4–4: Exceptional Function Returns

Function	Return Values
`sqlalloc()`	A pointer to an SQLVAR_T structure.
`sqlald()`	A pointer to an SQLDA_T structure.
`sqlcdbind()`	Only returns 0, -2, or -3. Never returns -1.
`sqldinit()`	No value returned (void).
`sqldsize()`	The number of bytes required to hold an SQLDA_T structure describing a specified number of columns (SQLVAR_T structures).
`sqlgetmsg()`	A pointer to the next message in the ESQL message queue.

No Data Outcome

The following functions can return the no data outcome. In this case, a value of 100 returns.

- `sqldynexec()`
- `sqldynrq()`
- `sqlrequest()`
- `sqlfetch()`
- `sqldynftch()`
- `sqldynposfetchh()`

These statements can also return the no data outcome.

- Singleton SELECT (SELECT INTO)

  The no data outcome returns if SELECT produces a result with 0 rows. No database values are assigned to the host variables. In this case, SELECT returns 100.

- Searched UPDATE

  The no data outcome returns if no row satisfies *search-condition*. In this case, UPDATE returns 100.

- Searched DELETE

  The no data outcome returns if *search-condition* is specified, but no row satisfies it. In this case, DELETE returns 100.

- INSERT

  The no data outcome returns if INSERT included a query specification that produced no rows. In this case, INSERT returns 100.

4.2 ESQL-LIB Reference Entries

The following reference entries describe all normalized functions with the phrase "returns normalized values." The return values for exceptional functions are described explicitly.

To check the value of SQLCODE or SQLSTATE for a particular ESQL-LIB function, you must bind SQLCODE or SQLSTATE using `sqlcdbind()` immediately prior to calling that function. Binding SQLCODE or SQLSTATE with `sqlcdbind()` remains in effect only for the following ESQL-LIB function call.

NOTE: All ESQL-LIB functions return control to your ESQL application, and none intentionally cause image exit. If a fatal Progress error does occur, your application obtains control, and ESQL-LIB accepts no further calls to functions except `sqlgetmsg()`.

sqlald() — Allocate Dynamic SQL Descriptor

Allocates memory for the SQL descriptor area, including the SQLDA_T structure and one SQLVAR_T structure for each column. Supports the
EXEC SQL ALLOCATE DESCRIPTOR... statement.

On successful completion, `sqlald()` returns a pointer to the allocated SQLDA_T structure; otherwise it returns a 0 pointer value. Since a failed sqllogin returns to the original state as if no `sqllogin()` had been executed, `sqlald()` fails after a failed `sqllogin()`. An `sqlald()` call succeeds only when the application successfully logs in.

SYNTAX

```
struct SQLDA_T *
sqlald ( Progress )
  int  ncols;    /* INPUT */
```

ncols

The maximum number of columns available for a dynamic cursor statement. The maximum value for `ncols` is 512.

EXAMPLE

This example allocates an SQL descriptor area for 10 columns, and assigns it to `psqlda:`

```
#include "proesql.h"
struct SQLDA_T *psqlda=(struct SQLDA_T *) 0;
            .
            .
            .
/* sqlald allocates storage for the SQLDA structure, including
   an instance of the SQLVAR structure for each column,
   up to the maximum requested. */

if ((psqlda = sqlald((int) 10)) == (struct SQLDA_T *) 0)
{
    printf ("Couldn't alloc memory for SQLDA with 10 columns.\n");
    exit(2);
}
```

SEE ALSO

sqldld() — Deallocate Dynamic Descriptor Area, `proesql.h`

sqlalloc() — Allocate Dynamic Memory

Allocates memory to hold the DATA (and INDICATOR) values for the SQLVAR_T component of the SQL descriptor area (SQLDA_T structure).

On successful completion, `sqlalloc()` returns a pointer to the allocated storage; otherwise it returns a 0 pointer value:

SYNTAX

```
char *
sqlalloc ( nbytes )
  int  nbytes;    /* INPUT */
```

nbytes

The number of bytes to allocate for the DATA or INDICATOR components for an SQLVAR_T structure of the SQLDA_T structure. For the DATA component, you typically use a value for `nbytes` based on the SQL data type of the column (the TYPE component of the column's SQLVAR_T structure). Include an extra byte for any NULL terminator or other extra characters that the data type uses. If you can set the column to the unknown value (i.e., the value of the `NULLABLE` member of the SQLVAR_T structure is 1), also allocate storage for the INDICATOR component with the length `sizeof` (long).

NOTES

- This function returns a pointer to the allocated memory.
- The `sqlalloc()` function is a portable memory allocator. You may use your own memory allocation routines instead.
- When allocating memory to hold the data values for column retrieval or parameter markers, the most effective technique is to obtain the column or parameter marker attribute information by calling `sqldyndesc()` (for columns) or `sqldyndescinp()` (for parameter markers).
- However you allocate storage, the precise allocation size depends on the data type of the column or parameter marker. Be sure to complete the allocation before retrieving columns or assigning parameter values to the SQLVAR_T structure.

EXAMPLE

In this example, the `for` statement assigns `psqlvar` to each column (SQLVAR_T structure) of an SQL descriptor area (assumed to be allocated contiguously). The body of the loop allocates alloclen bytes to each data element, set according to the data type of the column. For a complete listing of this example, see the `alloc_disp_cols()` function in the `dyndemo.cc` sample ESQL source file provided with your installation:

```
#include "proesql.h"
struct SQLDA_T*psqlda;
            .
            .
            .
struct SQLVAR_T*psqlvar;
            .
            .
            .
for (psqlvar = psqlda->SQLVAR[0]; . . . ; . . . psqlvar++)
{
            .
            .
    psqlvar->DATA = (char *) sqlalloc(alloclen);

    /* If you expect any NULL values, you must set the INDICATOR
       pointer here. Memory is allocated for a C long for each
       column that can be NULL. */

    if (psqlvar->NULLABLE)
        psqlvar->INDICATOR = (long *)sqlalloc(sizeof(long));
            .
            .
            .
}
```

SEE ALSO

sqlfree() — Deallocate Dynamic Memory, `proesql.h`

sqlapplname() — Set the ESQL Application Name

Sets the name of your ESQL application from within your client program. ESQL uses the first two characters of this name to generate filenames for any SQL r-code files compiled from this point on during application execution.

This function returns normalized values. For more information, see the function return values in the "General Diagnostics" section in this chapter.:

SYNTAX

```
int
sqlapplname ( applname, namlen )
  char  *applname;    /* INPUT */
  int    namlen;       /* INPUT */
```

applname

A pointer to a string that contains the application name (limit of 12 characters). ESQL uses the first two characters of applname as the first two characters of the application's r-code filenames. If you specify a null string for this value, ESQL generates these filenames entirely as random hexadecimal number strings (plus the .r extension). Use the manifest constant ESQL_MAXAPPLNAME to allocate space for the application name.

namlen

The length of the string that applname specifies.

NOTES

- You might use this function to associate r-code files with different sections of your program. Otherwise, you can specify a single two-character filename prefix for all r-code that your application generates using the `sqllogin()` function.
- If you preprocess your application and specify the -a startup parameter to the SQL preprocessor, set the same application name using either this function or the `sqllogin()` function.

EXAMPLE

This example sets the name of the application from p1phase to p2phase during execution:

```
main()
{
   int retcode;
   retcode=sqllogin(0,(char **)0,"-PF application.pf","p1phase");
   /* SQL statements produce r-code filenames prefixed with "p1" */
   if(retcode == -1)
   {
       exit(2);
   }
            .
            .
            .
   sqlapplname("p2phase",7);
   /* SQL statements produce r-code filenames prefixed with "p2" */
            .
            .
            .
}
```

SEE ALSO

sqllogin() — Start a Progress ESQL Session

sqlcdbind() — Bind Your Error Code Address to ESQL

Binds the address of your local status variable, SQLCODE or SQLSTATE, to ESQL. This function supports the return of error and status codes from ESQL to your application after executing each SQL statement.

This function always returns 0 for a successful completion:

SYNTAX

```
int
sqlcdbind ( psqlcode, psqlstate)
  long  *psqlcode;  /* INPUT */
  char  *psqlstate;  /* INPUT */
```

psqlcode

The address of your SQLCODE variable, declared to be a long C integer.

psqlstate

The address of your SQLSTATE variable.

NOTES

- The preprocessor generates a call to `sqlcdbind()` to bind the SQLCODE or SQLSTATE address to ESQL-LIB.
- The SQL preprocessor does not check to ensure that SQLCODE or SQLSTATE is properly scoped according to C language conventions. You may declare SQLCODE or SQLSTATE external to all C functions or within a specific C function.
- Binding SQLCODE or SQLSTATE remains in effect only for the duration of the next ESQL-LIB function call that affects SQLCODE or SQLSTATE. Once that function returns, the binding is lost. To use the SQLCODE or SQLSTATE value for subsequent ESQL-LIB function calls, rebind SQLCODE or SQLSTATE before each call.

EXAMPLE

The preprocessor inserts both the call to `sqlcdbind()` and the test for SQLCODE < 0 as directed by a prior WHENEVER statement for each subsequent SQL statement:

```
/* EXEC SQL WHENEVER SQLERROR GOTO err; */
            .
            .
            .
/* EXEC SQL close x; */
{
    sqlcdbind(&SQLCODE, (char *)0);
    sqldynclose(sql0, SQL_STMTDROP);
    if (SQLCODE < 0) goto err;
}
```

SEE ALSO

Chapter 2, "Programming with ESQL," for more information on SQLCODE, SQLSTATE, and the WHENEVER statement.

sqlcommit() — Commit the Database Transaction

Commits the current database transaction. This function supports the EXEC SQL COMMIT WORK statement to commit any database transaction that is active since the most recent:

- Call to `sqllogin()`.
- Last COMMIT WORK statement.
- Last ROLLBACK WORK statement.

This function returns normalized values. For more information, see the function return values in the "General Diagnostics" section in this chapter:

SYNTAX

```
int
sqlcommit ()
```

NOTES

- When using the ESQL-LIB interface, there is always a database transaction active.
- Work done while a database is connected is not committed when you call `sqldiscon()`. You must commit work using the COMMIT WORK statement or a call to `sqlcommit()` before you disconnect the database, or before exiting from your program. Otherwise, the disconnection does not occur and your open transaction is rolled back. Databases do not disconnect until a transaction ends. Therefore, any disconnections pend until the application encounters a COMMIT WORK or ROLLBACK WORK statement or an `sqllogout()` call (which automatically rolls back all work). If the transaction does not complete and you reconnect the database, the disconnect never actually occurs.

EXAMPLE

This example determines from a flag (`dynpending`) that there is uncommitted work prior to an attempt to exit the application through a `switch` statement. It queries the user and commits the work as directed:

```
if (dynpending) /* There's uncommitted work outstanding */
{
    printf("You have uncommitted changes. Commit before exit? (Y/N)\n");
    gets(ans);
    if (*ans == 'y' || *ans == 'Y')
        /* EXEC SQL COMMIT WORK; */
        sqlcommit();
}
          .
          .
          .
```

SEE ALSO

sqlrollback() — Rollback Any Pending Transaction

sqlconn() — Connect Database(s)

Connects one or more databases to your program.

This function returns normalized values. For more information, see the function return values in the "General Diagnostics" section in this chapter:

SYNTAX

```
int
sqlconn ( pparms )
  char  *pparms;  /* INPUT */
```

pparms

A pointer to a string that contains database connection parameters. This string may also specify the Parameter File (-pf) startup parameter.

NOTES

- This function accepts database connection parameters. Any file to which the Parameter File (-pf) startup parameter refers must also contain only this subset.
- The `sqllogin()` function accepts both ESQL and Progress database connection parameters. However, because a database connection failure during login prevents all further access to ESQL-LIB, using `sqlconn()` for all database connections is more efficient.

EXAMPLE

This example connects database testdb in single-user mode, and assigns it the logical database name `dbase1`.:

```
sqlconn("-1 testdb -ld dbase1");
```

SEE ALSO

sqldiscon() — Disconnect a Database, sqllogin() — Start a Progress ESQL Session, *Progress Startup Command and Parameter Reference* , and the CONNECT statement in the *Progress Language Reference* for more information on the parameters that `sqlconn()` accepts.

sqldelrow() — Delete Row at Open Cursor

Deletes the row at the current position of an open cursor. This function supports the static EXEC SQL DELETE FROM...WHERE CURRENT OF... (positioned DELETE) statement. If you precompile the statement, the preprocessor generates the r-code file to run it.

This function returns normalized values. For more information, see the function return values in the "General Diagnostics" section in this chapter:

SYNTAX

```
int
sqldelrow ( sqln, flag, prcfile )
  SQLRQHDL  *sqln;     /* INPUT */
  int       flag;     /* INPUT */
  char       *prcfile;  /* INPUT */
```

sqln

A pointer to the request handle for the open cursor. For more information on request handles, see the "SQL Request Handles" section.

flag

An execution flag that ESQL initializes.

prcfile

A pointer to the name of the precompiled r-code file. If the statement is not precompiled, the argument resolves to a null string.

EXAMPLE

This example deletes the row at the current position of cursor x (associated with request handle sql0):

```
#include "proesql.h"
static SQLRQHDL *sql0 = (SQLRQHDL *) 0;
            .
            .
            .
/* EXEC SQL delete from customer where current of x; */
{
    sqlcdbind(&SQLCODE, (char *)0);
    sqldelrow(sql0,(int)0x010,"");
    if (SQLCODE < 0) goto err;
}
```

SEE ALSO

sqldynrq() — Execute a Dynamic Non-SELECT Statement for executing dynamic ESQL positioned DELETE statements, sqlrequest() — Execute a Static ESQL Statement for executing non-positional static ESQL statements, sqlupdrow() — Update Row at Open Cursor for executing static positioned UPDATE statements.

sqldinit() — Initialize an Allocated SQL Descriptor Area

Initializes an SQLDA_T structure and its associated SQLVAR_T structures (SQL descriptor area) for the specified number of columns.

This function returns no values:

SYNTAX

```
void
sqldinit ( psqlda, ncols )
  struct SQLDA_T  *psqlda;    /* INPUT-OUTPUT */
  int              ncols;     /* INPUT */
```

psqlda

A pointer to a SQLDA_T structure previously allocated by `sqlald()` or your own memory allocation routine.

ncols

The number of columns to initialize in `psqlda`.

NOTES

- The SQL descriptor area that `psqlda` specifies must already be allocated.
- This function resets the specified SQL descriptor area to the number of columns specified by `ncols` and initializes the structure members to zero and NULL values. It does not add or subtract to the existing memory allocated for the descriptor area.
- The SQLDA_T and SQLVAR_T structures to which `psqlda` refers must be contiguous with each other. In other words, `psqlda` must point to one SQLDA_T structure followed by `ncols` SQLVAR_T structures. The number of bytes allocated at `psqlda` must be greater than or equal to the number of bytes that `sqldsize(ncols)` returns.
- This function sets the DATA and INDICATOR members of SQLVAR_T structures to NULL.

CAUTION: Specifying an unallocated or noncontiguous SQL descriptor area, or setting `ncols` greater than the number of columns currently allocated for the SQL descriptor area can result in unpredictable, platform-specific behavior.

EXAMPLE

This example uses the `sqldsize()` function to obtain the number of bytes needed to store an SQL descriptor area for `nres` columns. It then uses the `malloc()` C run-time library routine to allocate the memory and passes the memory to `sqldinit()` to initialize the structures:

```
#include "proesql.h"
struct SQLDA_T *psqlda;
int        nres, nbytes;
             .
             .
             .
nres = 10;
nbytes = sqldsize(nres);

psqlda = (struct SQLDA_T *) malloc(nbytes);
if (psqlda == (struct SQLDA_T *) NULL)
{
    printf( "allocation error\n" );
    return;
}
sqldinit( psqlda, nres );
             .
             .
             .
```

SEE ALSO

sqlald() — Allocate Dynamic SQL Descriptor

sqldiscon() — Disconnect a Database

Disconnects a single database from your program.

This function returns normalized values. For more information, see the function return values in the "General Diagnostics" section in this chapter:

SYNTAX

```
int
sqldiscon  ( pldbname )
  char  *pldbname;    /* INPUT */
```

pldbname

Pointer to the logical name of the connected database.

NOTE

- Work done while a database is connected is not committed when you call `sqldiscon()`. You must commit work using the COMMIT WORK statement or a call to `sqlcommit()` before you disconnect the database, or before exiting from your program. Otherwise, the disconnection does not occur and your open transaction is rolled back. Databases do not disconnect until a transaction ends. Therefore, any disconnections pend until the application encounters a COMMIT WORK or ROLLBACK WORK statement or an `sqllogout()` call (which automatically rolls back all work). If the transaction does not complete and you reconnect the database, the disconnect never actually occurs.

EXAMPLE

This example disconnects the `dbase1` database:

```
sqldiscon("dbase1");
```

SEE ALSO

sqlconn() — Connect Database(s), the DISCONNECT statement in the *Progress Language Reference* for more information on disconnecting from a database.

sqldld() — Deallocate Dynamic Descriptor Area

Deallocates the memory for an SQL descriptor area, including the SQLDA_T structure and associated SQLVAR_T structures that `sqlald()` previously allocates. Supports the EXEC SQL DEALLOCATE DESCRIPTOR... statement.

This function returns normalized values. For more information, see the function return values in the "General Diagnostics" section in this chapter:

SYNTAX

```
int
sqldld ( psqlda )
  struct SQLDA_T    *psqlda;    /* INPUT-OUTPUT */
```

psqlda

A pointer to a SQLDA_T structure that `sqlald()` previously allocates.

NOTES

- Before calling `sqldld()`, free the memory that was used for data and indicator storage in the SQL descriptor area.
- You only need to call this function if you are done with the allocated storage and want to continue with other work.

CAUTION: Be sure to deallocate all allocated memory referenced by SQLVAR_T structures in the SQL descriptor area before you deallocate the descriptor area. Otherwise, the memory pointers (DATA and INDICATOR) are lost and the memory cannot be freed.

Do not deallocate the same SQL descriptor area more than once. This can cause a memory violation. Do not attempt to call `sqldld()` with any value for `psqlda` other than those a previous call to `sqlald()` returns. This causes unpredictable behavior.

EXAMPLE

This example deallocates memory for the SQL descriptor area to which ptable refers, but only if `ptable` points somewhere (is non-null):

```
#include "proesql.h"
struct SQLDA_T *ptable;
............
............
if (ptable) sqldld(ptable;
............
```

SEE ALSO

sqlald() — Allocate Dynamic SQL Descriptor, sqlalloc() — Allocate Dynamic Memory, sqlfree() — Deallocate Dynamic Memory, `proesql.h`

sqldsize() — Calculate the Size of an SQL Descriptor Area

Returns the number of bytes required to hold an SQL descriptor area for the specified number of columns.

This function returns a value specifying the number of bytes required:

SYNTAX

```
int
sqldsize ( ncols )
  int    ncols;    /* INPUT */
```

ncols

The number of columns in an SQL descriptor area.

NOTE

- This function calculates the combined size of the SQLDA_T and associated SQLVAR_T structures required to allocate the specified SQL descriptor area. It does not include in this calculation the number of bytes needed for DATA and INDICATOR values.

EXAMPLE

This example uses the `sqldsize()` function to obtain the number of bytes needed to store an SQL descriptor area for nres columns. It then uses the `malloc()` C run-time library routine to allocate the memory and passes the memory to `sqldinit()` to initialize the structures:

```
#include "proesql.h"
struct SQLDA_T *psqlda;
int        nres, nbytes;
             .
             .
             .
nres = 10;
nbytes = sqldsize(nres);

psqlda = (struct SQLDA_T *) malloc(nbytes);
if (psqlda == (struct SQLDA_T *) NULL)
{
    printf( "allocation error\n" );
    return;
}
sqldinit( psqlda, nres );
             .
             .
             .
```

SEE ALSO

sqlald() — Allocate Dynamic SQL Descriptor, sqldinit() — Initialize an Allocated SQL Descriptor Area

sqldynclose() — Close an SQL Statement

Closes a static cursor SELECT or prepared dynamic (cursor or non-cursor) statement. This function supports the EXEC SQL CLOSE... statement.

This function returns normalized values. For more information, see the function return values in the "General Diagnostics" section in this chapter:

SYNTAX

```
int
sqldynclose ( sqln, option )
  SQLRQHDL    *sqln;  /* INPUT */
  int          option;  /* INPUT */
```

sqln

A pointer to the request handle that a previous call to `sqlrequest()` (static), `sqldynprep()` (dynamic), or `sqldynrq()` (dynamic) returns. For more information on request handles, see the "SQL Request Handles" section.

option

The value of SQL_STMTCLS or SQL_STMTDROP, as defined in the `proesql.h` header file. Setting option to SQL_STMTCLS closes the cursor associated with `sqln` (if defined) and discards all pending rows, resetting the statement for repeated execution. Setting option to SQL_STMTDROP releases `sqln`, frees all resources associated with it, closes the associated cursor (if defined), and discards all pending rows, removing the statement from the ESQL context.

NOTES

- The Progress/ESQL EXEC SQL CLOSE... statement always specifies the SQL_STMTCLS option. To specify the SQL_STMTDROP option, execute `sqldynclose()` directly.
- If the statement is a static cursor SELECT statement (cursor specification), `sqldynclose()` always closes the statement according to the SQL_STMTDROP option. The option is ignored.

- If the statement is a prepared dynamic ESQL statement or cursor specification, using the SQL_STMTCLS option lets you re-execute the statement associated with `sqln` by calling `sqldynexec()` (possibly with a new set of parameter marker values). This option is appropriate when executing an SQL statement many times. You can prepare the statement once, and execute it many times (possibly for each new set of parameter marker values). After the last execution, you can close the statement by specifying `sqldynclose()` with the SQL_STMTDROP option.

- Using the SQL_STMTDROP option terminates all access to `sqln`. To re-execute the statement or cursor specification associated with `sqln`, prepare it again with a call to `sqldynprep()` or invoke the statement with another EXEC SQL PREPARE... statement.

- If you prepare a previously prepared dynamic ESQL statement or re-execute a static cursor SELECT statement without closing it, the previous instance of the statement is automatically closed according to the SQL_STMTDROP option.

EXAMPLE

This example prepares a dynamic cursor specification and later closes the statement (identified by `sql0`), as well as the cursor associated with it:

```
#include "proesql.h"
             .
             .
             .
long SQLCODE;
SQLRQHDL *sql0 = (SQLRQHDL *)0;
             .
             .
             .
sqlcdbind(&SQLCODE, (char *)0);
sqldynprep(&sql0, "SELECT * FROM customer", "STMT5");
if (SQLCODE < 0)
  return;
             .
             .
             .
sqlcdbind(&SQLCODE, (char *)0);
sqldynclose(sql0, SQL_STMTDROP);
if (SQLCODE < 0)
{
  printf("Error closing cursor statement\n");
  return;
}
```

SEE ALSO

`proesql.h`, sqldynexec() — Execute a Dynamic SQL Statement, sqldynprep() — Prepare a Dynamic SQL Statement, sqldynrq() — Execute a Dynamic Non-SELECT Statement, sqlrequest() — Execute a Static ESQL Statement

sqldyndesc() — Get Dynamic Column Descriptions

Gets column description information for a dynamic ESQL statement. This function supports the dynamic EXEC SQL DESCRIBE... statement.

This function returns normalized values. For more information, see the function return values in the "General Diagnostics" section in this chapter:

SYNTAX

```
int
sqldyndesc ( sqln, psqlda )
  SQLRQHDL  *sqln;  /* INPUT */
  SQLDA_T    *psqlda;    /* OUTPUT */
```

sqln

A pointer to the request handle that a previous call to `sqldynprep()` returns. For more information on request handles, see the "SQL Request Handles" section.

psqlda

A pointer to an SQLDA_T structure to receive the column description information for the columns in the result set that the statement associated with `sqln` specifies.

NOTES

- Typically, an application calls `sqldynprep()` to prepare a dynamic SQL statement, allocates the SQLDA_T structure, and calls `sqldyndesc()` to fill in the column descriptions. Before calling `sqldyndesc()`, the application can also call `sqldynncols()` to return the number of columns required for the result set and thus ensure that the SQLDA_T structure is large enough to hold the returned information.
- To receive attribute information, the supplied SQLDA_T structure must be allocated large enough to hold the number of columns in the result set (`psqlda->`SQLN is equal to or greater than the number of columns in the result set). If it is not large enough, `sqldyndesc()` sets `psqlda->`COLCNT to the actual number of columns required and returns 0 with no attribute information in the structure.

EXAMPLE

This example describes and initializes the column information in the SQL descriptor area to which `psqlda` refers for a cursor specification:

```
#include "proesql.h"
static SQLRQHDL *sql1 = (SQLRQHDL *) 0;
static long SQLCODE;
struct SQLDA_T *psqlda = (struct SQLDA_T *) 0;
..........
..........
..........
if ((psqlda = sqlald((int) 10)) == (struct SQLDA_T *) 0)
{
  printf("Couldn't alloc memory for SQLDA with 10 columns.\n");
  break;
}
...........
...........
...........
sqlcdbind(&SQLCODE, (char *)0);
sqldynprep(&sql1, "SELECT * FROM customer", "DEMOSTMT");
...........
...........
...........
/* EXEC SQL DESCRIBE DEMOSTMT USING SQL DESCRIPTOR psqlda; *
sqlcdbind(&SQLCODE, (char *)0);
sqldyndesc(sql1, psqlda);
```

SEE ALSO

sqlald() — Allocate Dynamic SQL Descriptor, sqldynncols() — Get the Number of Columns in a Result Set, sqldynprep() — Prepare a Dynamic SQL Statement, `proesql.h`

sqldyndescinp() — Get Dynamic Parameter Descriptions

Gets description information for all parameter markers in a dynamic ESQL statement.

This function returns normalized values. For more information, see the function return values in the "General Diagnostics" section in this chapter:

SYNTAX

```
int
sqldyndescinp ( sqln, psqlda )
  SQLRQHDL    *sqln;  /* INPUT */
  SQLDA_T      *psqlda;  /* OUTPUT */
```

sqln

A pointer to the request handle that a previous call to `sqldynprep()` returns. For more information on request handles, see the "SQL Request Handles" section.

psqlda

A pointer to an SQLDA_T structure to receive the column description information for any parameter markers that the statement associated with `sqln` specifies.

NOTES

- Typically, an application calls `sqldynprep()` to prepare a dynamic SQL statement, allocates an SQLDA_T structure, and calls `sqldyndescinp()` to fill in the parameter marker descriptions. Before calling `sqldyndescinp()`, the application can also call `sqldynnparms()` to return the number of parameter markers in the statement, and use that information to allocate an SQLDA_T structure large enough to hold the returned information.
- To receive dynamic parameter marker description information, allocate the supplied SQLDA_T structure large enough to hold the number of parameter markers in the statement (`psqlda->`SQLN is equal to or greater than the number of parameters in the statement). If it is not large enough, `sqldyndescinp()` sets `psqlda->`COLCNT to the actual number of parameter markers and returns 0 with no parameter description information in the structure.

EXAMPLE

This example describes and initializes the parameter marker attribute information in the SQL descriptor area to which `psqlda` refers for a prepared searched UPDATE:

```
#include "proesql.h"
static SQLRQHDL *sql1 = (SQLRQHDL *) 0;
static long    SQLCODE;
struct SQLDA_T   *psqlda = (struct SQLDA_T *) 0;
            .
            .
            .
if ((psqlda = sqlald((int) 2)) == (struct SQLDA_T *) 0)
{
  printf("Couldn't alloc memory for 2 parameter markers.\n");
  break;
}
            .
            .
            .
sqlcdbind(&SQLCODE, (char *)0);
sqldynprep(&sql1,
        "UPDATE customer SET name = ? WHERE cust\-num = ?,
        "STMT6"
      );
            .
            .
            .
sqlcdbind(&SQLCODE, (char *)0);
sqldyndescinp(sql1, psqlda);
```

SEE ALSO

sqlald() — Allocate Dynamic SQL Descriptor, sqldynnparms() — Get the Number of Parameter Markers, sqldynprep() — Prepare a Dynamic SQL Statement, `proesql.h`

sqldynexec() — Execute a Dynamic SQL Statement

Executes a prepared dynamic ESQL statement. This function supports both the dynamic EXEC SQL OPEN... and EXEC SQL EXECUTE... statements.

This function returns normalized values. For more information, see the function return values in the "General Diagnostics" section in this chapter:

SYNTAX

```
int
sqldynexec ( sqln, psqlda )
  SQLRQHDL    *sqln;    /* INPUT */
  SQLDA_T      *psqlda;    /* OUTPUT */
```

sqln

A pointer to the request handle that a previous call to `sqldynprep()` returns. For more information on request handles, see the "SQL Request Handles" section.

psqlda

A pointer to an SQLDA_T structure that contains the data values for any dynamic parameter markers in the statement.

NOTES

- This function executes a statement prepared by a call to `sqldynprep()`. If there are parameter markers in the statement, supply the data values in the `psqlda` argument.
- If the prepared SQL statement is not a cursor specification, you can call `sqldynexec()` repeatedly to execute the statement multiple times, supplying different values for parameter markers if any are present. If the prepared SQL statement is a cursor specification, you can repeatedly execute the statement, but you must call `sqldynclose(sqln, SQL_STMTCLS)` before each call to `sqldynexec()`.

EXAMPLE

This example prepares and executes a cursor specification without any parameter markers:

```
#include "proesql.h"
static SQLRQHDL *sql0 = (SQLRQHDL *) 0;
static long    SQLCODE;
            .
            .
            .
sqlcdbind(&SQLCODE, (char *)0);
sqldynprep(&sql0, "SELECT * FROM customer", "STMT5");
if (SQLCODE < 0)
  return;
            .
            .
            .
sqlcdbind(&SQLCODE, (char *)0);
sqldynexec(sql0, (struct SQLDA_T *)0);
if (SQLCODE < 0)
{
  printf("Error executing request\n");
  sqldynclose(sql0, SQL_STMTDROP);
  return;
}
```

SEE ALSO

sqldynclose() — Close an SQL Statement, sqldynprep() — Prepare a Dynamic SQL Statement, and sqlrequest() — Execute a Static ESQL Statement for executing static ESQL statements.

sqldynftch() — Fetch Next Row of Dynamic Cursor

Fetches the next row of a dynamic open cursor. This function supports the dynamic EXEC SQL FETCH... statement sequence.

This function returns normalized values. For more information, see the function return values in the "General Diagnostics" section in this chapter:

SYNTAX

```
int
sqldynftch ( sqln, psqlda )
  SQLRQHDL        *sqln;      /* INPUT */
  struct SQLDA_T    *psqlda;      /* OUTPUT */
```

sqln

A pointer to the request handle for the open cursor. For more information on request handles, see the "SQL Request Handles" section.

psqlda

The pointer to the SQL descriptor area to receive the column data values. Be sure to allocate data storage for each column in the descriptor area.

NOTE

- If you fetch with host variables and use the preprocessor, the preprocessor declares its own `psqlda` to reference your host variables. If you do not use the preprocessor, initialize your own `psqlda` to reference the host variables in your FETCH statement.

EXAMPLE

This example fetches the next row of cursor `C1` (associated with a cursor specification prepared and opened using request handle `sql1`) into the SQL descriptor area to which `psqlda` points:

```
#include "proesql.h"
static SQLRQHDL *sql1 = (SQLRQHDL *) 0;
            .
            .
            .
/* EXEC SQL FETCH C1 INTO DESCRIPTOR psqlda; */
{
    sqlcdbind(&SQLCODE, (char *)0);
    sqldynftch(sql1,psqlda);
}
if (SQLCODE == 100) break;   /* NOT FOUND means End of Data */
```

SEE ALSO

sqlfetch() — Fetch Next Row of Open Cursor for static fetches, sqldynposfetch() — Fetch Row of Dynamic Open Cursor for positioned fetches.

sqldyngetn() — Get a Cursor Name

Gets the cursor name currently associated with the prepared dynamic ESQL statement that a request handle specifies.

This function returns normalized values. For more information, see the function return values in the "General Diagnostics" section in this chapter:

SYNTAX

```
int
sqldyngetn ( sqln, pname )
  SQLRQHDL    *sqln;    /* INPUT */
  char         *pname;      /* OUTPUT */
```

sqln

A pointer to the request handle that a previous call to `sqldynprep()` returns. For more information on request handles, see the "SQL Request Handles" section.

pname

The address of a memory location to receive the cursor name. The memory must be at least ESQL_MAXCURSNAME + 1 bytes long (the extra byte for the NULL terminator).

NOTES

- The SQL statements that use a cursor name are FETCH and positioned UPDATE and DELETE statements.
- All cursor names within an ESQL session are unique. If the application does not define a cursor name with a call to `sqldynsetn()`, ESQL generates one automatically when `sqldynprep()` prepares a SELECT statement.

EXAMPLE

This example prepares a cursor specification and declares a cursor for it using the DECLARE statement. After opening the cursor and fetching a row (not shown), the example retrieves the name of the cursor in `cursorname` and uses it in a positioned UPDATE request that sets the customer name in the current row to NULL:

```
#include "proesql.h"
static SQLRQHDL *sql0 = (SQLRQHDL *) 0;
static SQLRQHDL *sql1 = (SQLRQHDL *) 0;
static long      SQLCODE;
char        request[80];
char        cursorname[ESQL_MAXCURSNAME + 1];
            .
            .
            .
sqlcdbind(&SQLCODE, (char *)0);
sqldynprep(&sql0, "SELECT * FROM customer", "STMT5");
if (SQLCODE < 0)
  return;

/* EXEC SQL DECLARE C1 CURSOR FOR STMT5; */
{
  sqlcdbind(&SQLCODE, (char *)0);
  sqldynsetn(sql0, "C1");
}
            .
            .
            .
sqlcdbind(&SQLCODE, (char *)0);
sqldyngetn(sql0, cursorname);
if (SQLCODE < 0)
{
  printf("Error getting cursor for request\n");
  sqldynclose(sql0, SQL_STMTDROP);
  return;
}

sprintf( request,
      "UPDATE customer SET name = NULL WHERE CURRENT OF %s",
      cursorname
    );
sqlcdbind(&SQLCODE, (char *)0);
sqldynrq(&sql1, request);
if (SQLCODE < 0)
{
  printf("Error executing immediate request\n");
  return;
}
```

SEE ALSO

sqldynexec() — Execute a Dynamic SQL Statement, sqldynsetn() — Define a Cursor Name, proesql.h

sqldynncols() — Get the Number of Columns in a Result Set

Gets the number of columns in a dynamic ESQL result set.

This function returns normalized values. For more information, see the function return values in the "General Diagnostics" section in this chapter:

SYNTAX

```
int
sqldynncols ( sqln, ncolumns )
  SQLRQHDL    *sqln;    /* INPUT */
  int          *ncolumns; /* OUTPUT */
```

sqln

A pointer to the request handle that a previous call to `sqldynprep()` returns. For more information on request handles, see the "SQL Request Handles" section.

ncolumns

A pointer to an integer to receive the number of columns in the result set.

NOTES

- If the statement associated with `sqln` does not return columns, `sqldynncols()` sets `ncolumns` to 0.
- You must call `sqldynprep()` prior to any call to `sqldynncols()`.

EXAMPLE

This example prepares a cursor specification and returns the number of columns in the result set to ncolumns:

```
#include "proesql.h"
static SQLRQHDL *sql0 = (SQLRQHDL *) 0;
static long      SQLCODE;

int          ncolumns;
            .
            .
            .
sqlcdbind(&SQLCODE, (char *)0);
sqldynprep(&sql0, "SELECT * FROM customer", "STMT5");
if (SQLCODE < 0)
  return;

sqlcdbind(&SQLCODE, (char *)0);
sqldynncols(sql0, &ncolumns);
if (SQLCODE < 0)
{
  printf("Error returning number of columns for SELECT\n");
  sqldynclose(sql0, SQL_STMTDROP);
  return;
}
```

SEE ALSO

sqldynnparms() — Get the Number of Parameter Markers, sqldynprep() — Prepare a Dynamic SQL Statement

sqldynnparms() — Get the Number of Parameter Markers

Gets the number of parameter markers in a prepared dynamic ESQL statement.

This function returns normalized values. For more information, see the function return values in the "General Diagnostics" section in this chapter:

SYNTAX

```
int
sqldynnparms ( sqln, nparms )
  SQLRQHDL    *sqln;     /* INPUT */
  int          *nparms;      /* OUTPUT */
```

sqln

A pointer to the request handle that a previous call to `sqldynprep()` returns. For more information on request handles, see the "SQL Request Handles" section.

nparms

A pointer to an integer to receive the number of parameter markers in the prepared statement.

NOTES

- If the statement associated with `sqln` does not contain parameter markers, `sqldynnparms()` sets nparms to 0.
- You must call `sqldynprep()` prior to any call to `sqldynnparms()`.
- If you are using the preprocessor and are preparing and executing SQL requests using dynamic ESQL statements, the request handles are not available at development time, only the statement identifiers. (The preprocessor generates the request handles.) To call `sqldynnparms()` in this case, first call `sqlgetrqhdl()` to obtain the request handles for prepared statements to which the statement identifier refers.

EXAMPLE

This example prepares a cursor specification and returns the number of dynamic parameter markers in the WHERE clause to `nparams`.

The parameter markers in the request are hard-coded to show their presence. In reality, the request string might be conditionally assembled from several code points, making the presence and number of parameter markers truly unknown when the request is prepared:

```
#include "proesql.h"
static SQLRQHDL  *sql0 = (SQLRQHDL *) 0;
static long      SQLCODE;

int          nparams;
             .
             .
             .
sqlcdbind(&SQLCODE, (char *)0);
sqldynprep(&sql0, "SELECT * FROM customer WHERE name = ?", "STMT5");
if (SQLCODE < 0)
  return;
             .
             .
             .
sqlcdbind(&SQLCODE, (char *)0);
sqldynnparms(sql0, &nparams);
if (SQLCODE < 0)
{
  rintf("Error returning number of parameter markers\n");
  sqldynclose(sql0, SQL_STMTDROP);
  return;
}
```

SEE ALSO

sqldynncols() — Get the Number of Columns in a Result Set, sqldynprep() — Prepare a Dynamic SQL Statement, sqlgetrqhdl() — Get a Dynamic Request Handle

sqldynposfetch() — Fetch Row of Dynamic Open Cursor

Fetches the specified row of a dynamic open cursor.

This function returns normalized values:

SYNTAX

```
int
sqldynposfetch ( sqln, orientation, psqlin, psqlda)
  SQLRQHDL    *sqln;    /* INPUT */
  int         orientation;    /* INPUT */
  struct    SQLVAR_T    *psqlin;    /* INPUT */
  struct     SQLDA_T      *psqlda;    /* OUTPUT */
```

sqln

A pointer to the request handle for the open cursor. For more information on request handles, see the "SQL Request Handles" section.

orientation

Specifies which row to fetch. Use one of the following values defined in `proesql.h`: ESQL_FETCH_NEXT, ESQL_FETCH_FIRST, ESQL_FETCH_LAST, ESQL_FETCH_PREV, ESQL_FETCH_ABSOLUTE, or ESQL_FETCH_RELATIVE.

You must supply a numeric value for ESQL_FETCH_ABSOLUTE and ESQL_FETCH_RELATIVE. ESQL_FETCH_ABSOLUTE uses a row number relative to the beginning of the cursor, beginning with 1. ESQL_FETCH_RELATIVE uses a positive or negative relative position, including 0, which re-fetches the same row.

psqlin

The integer that specifies the row. This is required only for ESQL_FETCH_ABSOLUTE and ESQL_FETCH_RELATIVE.

psqlda

The pointer to the SQL descriptor area that receives the column data values. Be sure to allocate data storage for each column in the descriptor area.

NOTES

- Currently, no embedded SQL statement uses this call.
- To use orientations other than ESQL_FETCH_NEXT, set ESQL_SCROLLABLE_STATIC with `sqlsetlogoption()` prior to preparing the SELECT statement with `sqldynprep()`.
- For a scrollable cursor to support positioned updates and deletes, correlated subqueries must not occur in the query (in addition to the normal restrictions on cursor updates).
- Fetch orientation, `psqlin` combinations that specify a row beyond the end or before the beginning of available rows will yield sqlcode = 100, and sqlstate = 02000.
- Once a row has been fetched or fetched past, and you re-fetch the same row later, it may or may not be sensitive to intervening updates or deletes. Do not assume that the row is sensitive, since sensitivity depends on many features of the query.
- If a cursor is sensitive to updates and deletes, each re-fetch of a row displays its most recent updated state. If a referenced table row has been deleted, then a sensitive cursor returns sqlcode = -1, sqlstate = 50000 on an attempt to fetch from that position. The `sqlgetmst()` call retrieves messages that indicate which table had a deletion that undermines the requested cursor position. In the case of a sensitive join query, any one of several tables could be involved.
- This routine fails to conform to the ANSI-SQL92 scrollable cursor behavior in one respect: negative absolute positions between -1 and -(cursor cardinality) are valid according to SQL92, and name the last row through the first row, respectively. The `sqldynposfetch()` call does not implement this feature. Negative absolute positions return sqlcode=100, sqlstate=50000, and you are left positioned before the first row.
- There are no restrictions on what queries may be used in scrollable cursors. Any supported SQL query may be scrolled if the cursor is prepared with ESQL_SCROLLABLE_STATIC.

EXAMPLE

This example executes an absolute fetch of the data in row 50:

```
#include "proesql.h"
static SQLRQHDL *sql1 = (SQLRQHDL *) 0;
SQLVAR_T  t_scroll_pos;
long    scroll_amt;

t_scroll_pos.data = &scroll_amt;
t_scroll_pos.indicator = NULL;
t_scroll_pos.type = SQLDAINT;
     .
     .
     .
/* EXEC SQL FETCH C1 INTO DESCRIPTOR psqlda; */

{
  scroll_amt = 50;
  sqlcdbind(&SQLCODE, (char *)0);
  sqldynposfetch(sql1,ESQL_FETCH_ABSOLUTE,&t_scroll_pos,psqlda);
}
if (SQLCODE == 100) break;   /* NOT FOUND means invalid position */
```

SEE ALSO

sqlfetch() — Fetch Next Row of Open Cursor for static fetches, sqldynftch() — Fetch Next Row of Dynamic Cursor for dynamic fetches, and sqlsetlogoption() — Set a Session-wide Option to enable/disable scrollable cursors.

sqldynprep() — Prepare a Dynamic SQL Statement

Prepare a dynamic SQL statement. This function supports the dynamic EXEC SQL PREPARE... statement.

This function returns normalized values. For more information, see the function return values in the "General Diagnostics" section in this chapter:

SYNTAX

```
int
sqldynprep ( sqln, prequest, pstmtid )
  SQLRQHDL     **sqln;        /* OUTPUT */
  char         *prequest;     /* INPUT */
  char         *pstmtid;     /* INPUT */
```

sqln

The address of a pointer to the request handle value associated with and returned for the statement being prepared. If the function returns successfully, you can use `sqln` for subsequent calls to ESQL-LIB to refer to this statement. For more information on request handles, see the "SQL Request Handles" section.

prequest

A pointer to the text of the SQL statement being prepared.

pstmtid

A pointer to a NULL-terminated string that contains the SQL statement identifier (limited to 18 characters). If you use the preprocessor, it supplies this value. Otherwise, it can be NULL. If you set it to NULL, `sqlgetrqhdl()` cannot return the request handle corresponding to the statement identifier.

NOTES

- This function prepares a dynamic ESQL statement for execution and associates the result with a request handle (`sqln`). ESQL-LIB makes all future references to this statement using `sqln`. For example, the application can execute and re-execute the prepared statement by calling `sqldynexec()` until the application frees `sqln` with a call to `sqldynclose(sqln`, SQL_STMTDROP) or until the application reinitializes `sqln` with a subsequent call to `sqldynprep()` or `sqldynrq()`.

- Once the statement is prepared, the application can request information about the statement with calls to `sqldynncols()`, `sqldynnparms()`, `sqldyndesc()`, or `sqldyndescinp()`.

- You can prepare a statement once and execute it repeatedly using `sqldynexec()` with new parameter marker values. For more information on using dynamic parameter markers, see Chapter 2, "Programming with ESQL."

- If you prepare a statement using a request handle currently associated with a previously prepared statement, the previous statement is dropped and any associated cursor is closed before preparing the specified statement.

- Dynamic request strings cannot contain the following:

  – DECLARE CURSOR, OPEN, FETCH, or CLOSE statements used to manage cursor specifications.

  – References to embedded host variable names.

  – SQL comments.

  – dynamic parameters in a context that prevents the data type of the surrounding expression from being inferred.

  For more information and additional restrictions in the request string, see Chapter 2, "Programming with ESQL."

EXAMPLE

This example prepares the dynamic ESQL cursor specification in the character string request:

```
#include "proesql.h"
static SQLRQHDL *sql1 = (SQLRQHDL *) 0;
            .
            .
            .
char *request= "SELECT * FROM customer";

/* EXEC SQL PREPARE DEMOSTMT FROM request; */

{
    sqlcdbind(&SQLCODE, (char *)0);
    sqldynprep(&sql1, request, "DEMOSTMT");
}
if (SQLCODE < 0)
{
  printf("Error preparing cursor SELECT\n");
  return;
}
```

SEE ALSO

sqldyndesc() — Get Dynamic Column Descriptions, sqldynexec() — Execute a Dynamic SQL Statement, sqldynncols() — Get the Number of Columns in a Result Set, sqldynnparms() — Get the Number of Parameter Markers

sqldynrq() — Execute a Dynamic Non-SELECT Statement

Compiles and executes any dynamic ESQL statement other than a cursor specification. This function supports the dynamic EXEC SQL EXECUTE IMMEDIATE... statement.

This function returns normalized values. For more information, see the function return values in the "General Diagnostics" section in this chapter:

SYNTAX

```
int
sqldynrq ( sqln, prequest )
  SQLRQHDL    **sqln;    /* OUTPUT */
  char         *prequest;  /* INPUT */
```

sqln

The address of a pointer to a request handle value associated with and returned for the statement. For more information on request handles, see the "SQL Request Handles" section.

prequest

A pointer to the text of a non-SELECT statement.

NOTES

- Examples of non-cursor specification statements are Data Definition Language (DDL) statements like CREATE TABLE or DROP TABLE, or simple Data Manipulation Language (DML) statements like INSERT or searched DELETE and UPDATE. You can also prepare (compile) and repeatedly execute any of these statements (possibly including parameter markers) using `sqldynprep()` and `sqldynexec()`.
- This function can also compile and execute positioned UPDATE and DELETE statements. The cursor name referenced in the statement must already be opened and positioned on a record.
- The `sqldynrq()` function is functionally equivalent to `sqldynprep()` followed by `sqldynexec()`.
- This function releases all memory for the SQL request after executing it.

- Dynamic request strings cannot contain the following:
  - DECLARE CURSOR, OPEN, FETCH, or CLOSE statements used to manage cursor specifications.
  - References to embedded host variable names.
  - SQL comments.
  - Dynamic parameters in a context that prevents the data type of the surrounding expression from being inferred.

  For more information and additional restrictions in the request string, see Chapter 2, "Programming with ESQL."

EXAMPLE

This example prompts the user to enter an UPDATE or DELETE statement for some table. The user can actually enter either a positioned or searched UPDATE or DELETE, depending on the availability of an open cursor. Statements are passed in the string request:

```
#include "proesql.h"
static SQLRQHDL *sql0 = (SQLRQHDL *) 0;
            .
            .
            .
printf("Enter 'update/delete <table> ... where ...' request:\n");
gets(request);
/* EXEC SQL EXECUTE IMMEDIATE statement; */
{
    sqlcdbind(&SQLCODE, (char *)0);
    sqldynrq(&sql0,request);
}
```

SEE ALSO

sqldynexec() — Execute a Dynamic SQL Statement, sqldynprep() — Prepare a Dynamic SQL Statement, sqlrequest() — Execute a Static ESQL Statement for static non-cursor statements.

sqldynsetn() — Define a Cursor Name

Defines and associates a cursor name with a prepared dynamic ESQL statement. This function supports the dynamic EXEC SQL DECLARE... statement.

This function returns normalized values. For more information, see the function return values in the "General Diagnostics" section in this chapter:

SYNTAX

```
int
sqldynsetn ( sqln, pname )
  SQLRQHDL    *sqln;    /* INPUT */
  char         *pname;      /* INPUT */
```

sqln

A pointer to the request handle that a previous call to `sqldynprep()` returns. For more information on request handles, see the "SQL Request Handles" section.

pname

A pointer to a NULL-terminated string that contains a cursor name (limited to 18 characters) to associate with the SQL statement to which `sqln` refers. If you use the preprocessor, it supplies this value.

NOTES

- The SQL statements that use a cursor name are FETCH and positioned UPDATE and DELETE statements:

```
EXEC SQL UPDATE INVTABLE ... WHERE CURRENT OF INVCURSOR;
```

- All cursor names within an ESQL session must be unique. If the application does not define a cursor name with a call to `sqldynsetn()`, ESQL generates one automatically when a SELECT statement is compiled.
- Character strings for cursor names are limited to ESQL_MAXCURSNAME characters (plus the NULL terminator), as defined in `proesql.h`. The cursor name must be less than or equal to ESQL_MAXCURSNAME + 1 characters in length.
- The application cannot modify a cursor name once that cursor is opened with a call to `sqldynexec()`.

EXAMPLE

This example prepares a cursor specification and declares a cursor for it using the DECLARE statement, converted to a call to `sqldynsetn()`. It also shows how you can use the declared cursor to execute a positioned UPDATE:

```
#include "proesql.h"
static SQLRQHDL *sql0 = (SQLRQHDL *) 0;
static SQLRQHDL *sql1 = (SQLRQHDL *) 0;
static long  SQLCODE;
char  cursorname[ESQL_MAXCURSNAME + 1];
char  *request="UPDATE customer SET name=NULL WHERE CURRENT OF C1";
            .
            .
            .
sqlcdbind(&SQLCODE, (char *)0);
sqldynprep(&sql0, "SELECT * FROM customer", "STMT5");
if (SQLCODE < 0)
  return;

/* EXEC SQL DECLARE C1 CURSOR FOR STMT5; */
{
  sqlcdbind(&SQLCODE, (char *)0);
  sqldynsetn(sql0, "C1");
}
if (SQLCODE < 0)
{
  printf("Error setting cursor for SELECT\n")
  return;
}
            .
            .
            .

sqlcdbind(&SQLCODE, (char *)0);
sqldynrq(&sql1, request);
```

SEE ALSO

sqldynexec() — Execute a Dynamic SQL Statement, sqldynprep() — Prepare a Dynamic SQL Statement, sqldyngetn() — Get a Cursor Name

sqlfetch() — Fetch Next Row of Open Cursor

Fetches the next row of data for an open cursor. This function supports the EXEC SQL FETCH...INTO... statement.

This function returns normalized values. For more information, see the function return values in the "General Diagnostics" section in this chapter:

SYNTAX

```
int
sqlfetch ( sqln, flag, prequest, pvformat, psqlv)
  SQLRQHDL     *sqln;     /* INPUT */
  int          flag;       /* INPUT */
  char         *prequest; /* INPUT */
  char         *pvformat; /* INPUT */
  char         *psqlv[];  /* INPUT */
```

sqln

A pointer to the request handle for the open cursor. For more information on request handles, see the "SQL Request Handles" section.

flag

An execution flag that the SQL preprocessor initializes.

prequest

A pointer to the text of the statement.

pvformat

A pointer to a format string that describes the types and lengths of the embedded host variables.

psqlv

A pointer to the host variables to which the current SQL request refers.

EXAMPLE

This example fetches the next row of the open cursor x (associated with request handle sql0), and deposits the column values into the corresponding local variables `custnum`, `nam`, `city`, `state`, and `maxcred`:

```
#include "proesql.h"
static SQLRQHDL *sql0 = (SQLRQHDL *) 0;
            .
            .
            .
    /* EXEC SQL BEGIN DECLARE SECTION; */
    char nam[30];
    long custnum, maxcred;
    char city[20], state[3];
    /* EXEC SQL END DECLARE SECTION; */
            .
            .
            .
    /* EXEC SQL fetch x into :custnum,
         :nam, :city, :state, :maxcred; */
    {
       static char sqltxt[] = {
       'f','e','t','c','h',',','x',',','i','n','t','o',',','e','s','q',
       'l','o','0',',',',',',','e','s','q','l','o','1',',',',',',','e','s','q',
       'l','o','2',',',',',',','e','s','q','l','o','3',',',',',',','e','s','q',
       'l','o','4','.',0};
       static char sqlfmt[] =
       {'4','l','3','0','s','2','0','s','3','s','4','l',0};
       static char *sqlv[6];
       sqlv[0] = (char *) &custnum;
       sqlv[1] = (char *) nam;
       sqlv[2] = (char *) city;
       sqlv[3] = (char *) state;
       sqlv[4] = (char *) &maxcred;
       sqlv[5] = (char *) 0;
       sqlcdbind(&SQLCODE, (char *)0);
       sqlfetch(sql0,(int) 0x08,sqltxt,sqlfmt,sqlv);
       if (SQLCODE == 100) goto done1;
       if (SQLCODE < 0) goto err;
            .
            .
    }
```

SEE ALSO

sqldynftch() — Fetch Next Row of Dynamic Cursor for dynamic fetches, sqlrequest() — Execute a Static ESQL Statement for opening static cursors, and sqldynposfetch() — Fetch Row of Dynamic Open Cursor for positioned fetches.

sqlfree() — Deallocate Dynamic Memory

Deallocates the memory for a column data component that `sqlalloc()` previously allocates.

This function returns normalized values. For more information, see the function return values in the "General Diagnostics" section in this chapter:

SYNTAX

```
int
sqlfree ( pmem )
  char  *pmem;  /* INPUT-OUTPUT */
```

pmem

A pointer to memory that `sqlalloc()` previously allocates.

NOTES

- Call `sqlfree()` for the data component of each column every time you close a cursor, and before you execute new queries that allocate storage.
- Always call this function before calling `sqldld()`.

CAUTION: Do not deallocate the same block of memory more than once. This can cause a memory violation. Do not attempt to call `sqlalloc()` with any value for pmem other than what `sqlalloc()` returns. This causes unpredictable behavior.

EXAMPLE

This example deallocates the memory for the data elements in a column to which psqlvar refers:

```
#include "proesql.h"
struct SQLVAR_T *psqlvar;
            .
            .
            .
sqlfree(psqlvar->DATA);
psqlvar->DATA = (char *) 0;
if (psqlvar->INDICATOR)
{
    /* Free storage for NULL value indicators if allocated. */
    sqlfree(psqlvar->INDICATOR);
    psqlvar->INDICATOR = (long *) 0;
}
```

SEE ALSO

sqlalloc() — Allocate Dynamic Memory, sqldld() — Deallocate Dynamic Descriptor Area, proesql.h

sqlgetmsg() — Get Next Error Message

Gets the next message in the Progress message queue.

This function returns a pointer to the next error message in the message queue. If there are no messages in the queue, it returns a NULL character pointer:

SYNTAX

```
char *
sqlgetmsg ()
```

NOTE

- The execution of each SQL statement may cause additional messages to be appended to the queue. This function provides one-way retrieval of these first-in-first-out messages.
- Once you retrieve a message, it is removed from the queue and you cannot retrieve it again using `sqlgetmsg()`. Your application must save any messages that it needs to retain.
- Progress clears the message queue at the start of each ESQL-LIB entry point except:
  - `sqlfree()`
  - `sqlgetmsg()`
  - `sqlalloc()`
  - `sqlcdbind()`
  - `sqldinit()`
  - `sqldld()`
  - `sqldsize()`

EXAMPLE

This example shows how to retrieve messages for a preceding SQL statement:

```
char *msg;
             .
             .
             .
/* SQL statement */
             .
             .
             .
while (msg = (char *) sqlgetmsg()) != (char *) 0)
    printf("%s\n",msg);
```

sqlgetrqhdl() — Get a Dynamic Request Handle

Gets a request handle corresponding to a prepared dynamic ESQL statement identifier.

This function returns normalized values. For more information, see the function return values in the "General Diagnostics" section in this chapter.

SYNTAX

```
int
sqlgetrqhdl ( pstmtid, sqln )
  char      *pstmtid;  /* INPUT */
  SQLRQHDL   **sqln;    /* OUTPUT */
```

pstmtid

A pointer to an SQL statement identifier used in dynamic ESQL statements and registered to the ESQL-LIB interface with a previous call to `sqldynprep()`.

sqln

The address of a pointer to the request handle corresponding to the specified statement identifier. For more information on request handles, see the "SQL Request Handles" section.

NOTE

- This function allows any ESQL application based on prepared dynamic ESQL statements (and thereby on statement identifiers) to access Progress ESQL functions (based on request handles) that retrieve and manipulate attribute information for table columns and dynamic parameter markers.

EXAMPLE

This example retrieves the request handle for a prepared cursor specification in order to get the number of parameter markers in the request. The example assumes that you have prepared the request using the PREPARE statement, before the preprocessor has defined and returned the request handle to `sql1`. However, the code itself is preprocessed, and that is why rqhandle, to which `sqlgetrqhdl()` returns the same request handle, appears redundant. If you call ESQL-LIB directly to implement dynamic SQL requests, you define and use your own request handle variables from the beginning and do not need to call `sqlgetrqhdl()`:

```
#include "proesql.h"
static SQLRQHDL *sql1 = (SQLRQHDL *) 0;
            .
            .
            .
SQLRQHDL  *rqhandle;
int    nparms;

char *request="SELECT * FROM customer WHERE credit-\limit > ?";

/* EXEC SQL PREPARE DEMOSTMT FROM request; */

{
  sqlcdbind(&SQLCODE, (char *)0);
  sqldynprep(&sql1, request, "DEMOSTMT");
}
if (SQLCODE < 0)
{
  printf("Error preparing cursor SELECT\n");
  return;
}
            .
            .
            .

sqlcdbind(&SQLCODE, (char *)0);
sqlgetrqhdl( "DEMOSTMT", &rqhandle );
if (SQLCODE < 0)
{
  printf("Error getting request handle\n");
  return;
}
sqlcdbind(&SQLCODE, (char *)0);
sqldynnparms(rqhandle, &nparms);
```

In this example, the parameter markers in the request are hard-coded to show their presence. In reality, the request string might be conditionally assembled from several code points, making the presence and number of parameter markers truly unknown when the request is prepared.

SEE ALSO

sqldynprep() — Prepare a Dynamic SQL Statement, sqldynnparms() — Get the Number of Parameter Markers

sqllogin() — Start a Progress ESQL Session

Logs into Progress ESQL from an ESQL application. You must successfully call `sqllogin()` once before calling any other ESQL functions, except `sqlgetmsg()`.

If `sqllogin()` logs in successfully and connects to all requested databases, it returns 0; otherwise it returns -1. If the call is unsuccessful, you can call `sqllogin()` again. (If you call `sqllogin()` more than once, it returns unsuccessfully after the first successful call.) You can also call `sqllogin()` again after you call `sqllogout()`:

SYNTAX

```
int
sqllogin ( argc, argv, pparms, papplname )
  int   argc;          /* INPUT */
  char  **argv;       /* INPUT */
  char  *pparms;       /* INPUT */
  char  * papplname;    /* INPUT */
```

argc

The number of pointers (to startup parameters) to which `argv` refers. If you set `argv` from the command line, the operating system typically sets `argc` when parsing the command line during application startup. However, you must ensure that `argc` is set according to the following rules:

- If `argv` points to one or more non-null pointers (i.e., there is a list of startup parameters), you must set `argc` to a value >= 1. It does not matter what value you choose (i.e., it need not match the number of pointers to which `argv` refers.
- If `argv` is null, (i.e., there is no list of startup parameters), you must set `argc` to zero.

argv

A pointer to a null-terminated list of pointers to Progress startup parameters. ESQL requires the null termination to find the end of the list (i.e., it does not use `argc` to traverse the list). If you set `argv` from the command line, the operating system terminates the list for you.

pparms

A pointer to a string specifying Progress startup parameters. ESQL processes the parameters in pparms before the parameters in `argv`. This allows you to override default values that `argv` specifies from the command line.

applname

A pointer to the name of your application. ESQL uses the first two characters of applname to generate the first two characters of the application's r-code filenames. If you specify a null string for this value, ESQL generates these filenames entirely as random hexadecimal number strings (plus the `.r` extension).

NOTE: If you preprocess with precompiling, the string that applname specifies must contain the same name as the application name you specify with the `-a` parameter of the SQL preprocessor.

NOTES

- You can pass Progress startup parameters to ESQL from the command line only using `sqllogin()`.
- You can call `sqllogin()` only once from an application.
- If `sqllogin()` fails, the only ESQL-LIB function you can call is `sqlgetmsg()` to read any messages that ESQL returns.
- The `sqllogin()` function accepts both ESQL session (login) startup parameters and database connection parameters. However, because a database connection failure during login prevents further access to ESQL-LIB (except `sqlgetmsg()`), you can build a more robust application using the `sqlconn()` function to make all database connections, and using `sqllogin()` for ESQL session startup only.
- If you are running in the small client model, you cannot pass Progress session-wide client startup parameters to `sqllogin()`, except for the common ESQL startup parameters. For more information, see Chapter 3, "Building and Running ESQL Applications."
- For another way to specify the application name, see the `sqlapplname()` function reference entry.
- For more information on r-code generation and logging in to a Progress database, see Chapter 2, "Programming with ESQL."

EXAMPLE

In this example, the program `esqldemo` calls `sqllogin()`:

```
char *msg;
            .
            .
            .
if (sqllogin(argc-1,&argv[1],"-s 10","esqldemo") == -1)
{
  printf("Failed to login to Progress ESQL!\n");
    while ((msg = (char *) sqlgetmsg()) != (char *) 0)
      printf("%s\n",msg);
  exit(2);
}
```

For this example, you must first invoke `esqldemo` with the following command:

```
esqldemo demo -U dean -P PdSvcs
```

The first command line argument (`argv`[0]) points to the program name esqldemo, but esqldemo bypasses `argv`[0], since Progress does not need or recognize the program name as a startup parameter. The first argument to `sqllogin()` (`argc`-1) passes a decremented argument count to reflect the bypassing of `argv`[0]. The program accepts the remaining command line arguments as Progress startup parameters (passed with the second argument to `sqllogin()` , `argv`[1]), including the database name (demo), the user name (dean), and the password (PdSvcs). The program passes the third argument to `sqllogin()` containing the Stack Size startup parameter (set to 10K), and passes the fourth argument to `sqllogin()` specifying `esqldemo` as the application name used for generating r-code.

SEE ALSO

sqlapplname() — Set the ESQL Application Name, sqlconn() — Connect Database(s), sqlgetmsg() — Get Next Error Message, sqllogout() — Terminate a Progress ESQL Session

sqllogout() — Terminate a Progress ESQL Session

Logs out of the ESQL session and returns control to the ESQL application.

This function returns normalized values. For more information, see the function return values in the "General Diagnostics" section in this chapter:

SYNTAX

```
int
sqllogout ( exitst )
  int    exitst;    /* INPUT */
```

exitst

This parameter is ignored.

NOTES

- If your application is logged in to Progress, `sqllogout()` logs out of Progress, terminating all connections to databases. If the last transaction is uncommitted, Progress rolls it back.
- Prior to Version 7, the Host Language Interface (HLI) redefined the standard C `exit(exitst)` function as `sqllogout(exitst)` because `sqllogout(exitst)` was designed to exit the application if it was passed a non-zero value in exitst. This redefinition of `exit(exitst)` no longer occurs. A call to `sqllogout(exitst)` no longer exits but returns to your application for any value of exitst. Therefore, you must follow each call to `sqllogout(exitst)` that you previously intended to exit with a call to `exit(exitst)`.

EXAMPLE

This example logs out of Progress, and returns control to the application:

```
main()
{
    .
    .
    .
 sqllogout(0);
}
```

SEE ALSO

sqllogin() — Start a Progress ESQL Session

sqlrequest() — Execute a Static ESQL Statement

Executes any static ESQL statement, except positioned UPDATE and DELETE. This function supports any non-positioned static ESQL statement or cursor specification (EXEC SQL DECLARE...CURSOR FOR SELECT and OPEN... statements). If you precompile the statement, the preprocessor generates the r-code file to run it.

This function returns normalized values. For more information, see the function return values in the "General Diagnostics" section in this chapter:

SYNTAX

```
int
sqlrequest ( sqln, flag, prequest, pcursnam, pvformat, psqlv)
  SQLRQHDL  **sqln;    /* OUTPUT */
  int       flag;      /* INPUT */
  char      *prequest;   /* INPUT */
  char      *pcursnam;   /* INPUT */
  char      *pvformat;   /* INPUT */
  char      *psqlv[];    /* INPUT */
```

sqln

The address of a pointer to a request handle value associated with and returned for the statement. If the statement is an open cursor statement, this argument tracks all statements for the cursor. For more information on request handles, see the "SQL Request Handles" section.

flag

An execution flag that ESQL initializes.

prequest

A pointer to the text of the statement. If you precompile the statement, the text contains the name of the r-code file associated with the statement.

pcursnam

A pointer to the name of the cursor for the statement. If it is a non-cursor request, the pointer resolves to a null string.

pvformat

A pointer to a format string that describes the types and lengths of the embedded host variables.

psqlv

A pointer to the host variables to which the current SQL request refers.

EXAMPLE

This example opens cursor x (associated with request handle sql0) for the customer table, and selects all rows of the specified columns, where column cust-num is greater than the local variable hicustnum:

```
#include "proesql.h"
static SQLRQHDL *sql0 = (SQLRQHDL *) 0;
            .
            .
            .
    /* EXEC SQL BEGIN DECLARE SECTION; */
    . . . long  . . . hicustnum; . . .
    /* EXEC SQL END DECLARE SECTION; */
    hicustnum = 7;
            .
            .
            .
    /* EXEC SQL declare x cursor for
        select cust\-num, name, city, st, max\-credit from customer
            where cust\-num < :hicustnum; */
    /* EXEC SQL OPEN x; */
    {
static char sqltxt[] = {
'd','e','c','l','a','r','e',',','x',',','c','u','r','s','o','r',' ',
'f','o','r',',','s','e','l','e','c','t',',','c','u','s','t','-',''n','u','m',
',',',','n','a','m','e',',',',','c','i','t','y',',',',' ',
's','t',',',',','m','a','x','-','c','r','e','d','i','t',',','f',''r','o','m',
',','c','u','s','t','o','m','e','r',',','w','h','e','r'','e',',',
'c','u','s','t','-','n','u','m',',','<',',','e','s','q','l'','i','0','.',0};
static char sqlfmt[] = {',','4','l',0};
static char *sqlv[2];
       sqlv[0] = (char *) &hicustnum;
       sqlv[1] = (char *) 0;
       sqlcdbind(&SQLCODE, (char *)0);
       sqlrequest(&sql0,(int)0x04,sqltxt,"x",sqlfmt, sqlv);
       if (SQLCODE < 0) goto err;
            .
            .
            .
    }
```

SEE ALSO

sqldelrow() — Delete Row at Open Cursor for executing positioned DELETE statements, sqldynexec() — Execute a Dynamic SQL Statement for executing dynamic requests, sqldynprep() — Prepare a Dynamic SQL Statement for compiling dynamic requests, sqldynrq() — Execute a Dynamic Non-SELECT Statement for compiling and executing dynamic non-SELECT statements, sqlfetch() — Fetch Next Row of Open Cursor for fetching rows of static cursors, sqlupdrow() — Update Row at Open Cursor for executing static positioned UPDATE statements.

sqlrollback() — Rollback Any Pending Transaction

Rolls back any pending database transaction. This function supports the EXEC SQL ROLLBACK WORK statement to cancel and roll back any database transaction that is active since the most recent:

- Call to `sqllogin()`.
- Last ROLLBACK WORK statement.
- Last COMMIT WORK statement.

This function returns normalized values. For more information, see the function return values in the "General Diagnostics" section in this chapter:

SYNTAX

```
int
sqlrollback ()
```

EXAMPLE

This example implements typical error processing code, displaying the latest SQL error messages (from `sqlgetmsg()`) and rolling back the incomplete transaction:

```
err:
  printf("Error occurred.\n");
  while ((msg = (char *) sqlgetmsg()) != (char *) 0)
      printf("%s\n",msg);
  printf("\nRolling back all work...\n");
  /* EXEC SQL ROLLBACK WORK; */
  sqlrollback();
          .
          .
          .
```

SEE ALSO

sqlcommit() — Commit the Database Transaction for committing transactions.

sqlsetlogoption() — Set a Session-wide Option

Sets a session-wide option to change subsequent run-time behavior of the ESQL interface.

This function returns 0 if it succeeds in setting the option. If it fails (for example, with an option value out of range), it returns -1:

SYNTAX

```
int
sqlsetlogoption ( option, param )
  int                option;   /* INPUT */
  unsigned long int    param    /* INPUT */
```

option

The value of ESQL_LOCKFUNC, ESQL_LOCKPARM, ESQL_SCROLLABLE_STATIC, or ESQL_FORWARD_ONLY, as defined in your `proesql.h` header file. The value of option determines the meaning of `param`. Other options may be defined in the future.

param

A value that corresponds to the value of *option*.

- ESQL_LOCKFUNC — *param* contains the address of your own callback function to affect ESQL behavior when it encounters a resource lock.
- ESQL_LOCKPARM — *param* contains a parameter value to pass to your resource lock callback function.
- _ESQL_FORWARD_ONLY — SELECT statements will be prepared as normal (not scrollable) cursors. This is the default following sqllogin. *param* is not used.
- ESQL_SCROLLABLE_STATIC — SELECT statements will be prepared as scrollable cursors. *param* is not used.

NOTES

- The ESQL_LOCKFUNC and ESQL_LOCKPARM options that this function supports affect the behavior of ESQL when a resource lock is encountered during a row fetch. It lets you specify a callback function that can respond to the lock.
- In future releases, other values for option may be supported.

- If you do not call `sqlsetlogoption()`, ESQL waits indefinitely for a locked resource. This is the default behavior.
- The interface for your callback function must conform to the following specification:

SYNTAX

```
int
callback ( param, puser, ptty )
  unsigned long int    param    /* INPUT */
  char                 *puser   /* INPUT */
  char                    *ptty   /* INPUT */
```

callback

Your callback function name.

param

The parameter setting in effect for the ESQL_LOCKPARM option at the time the SQL statement was prepared or opened.

puser

User name of the resource lock owner that ESQL passes.

ptty

The terminal (TTY) name of the resource lock owner that ESQL passes.

The *callback* function cannot perform any database operations and must return one of the values listed in Table 4–5.

Table 4–5: Lock Callback Function Return Codes

Return Code	Description
-2	Wait for the lock to clear indefinitely, and make no further callbacks to the function (default behavior).
-1	Continue waiting for the lock to clear, but make another callback to the function.
0	Stop waiting for the lock to clear and return to the application with SQLCODE set to -1 (error).

- A common use of the callback function is to display a message that contains the lock owner and the terminal name in response to the lock condition. It then returns with a required value.

- Setting the callback function is on a per-request basis. Each setting is non-retroactive to any pending requests that have already been prepared or executed. For example, suppose you have not yet established a lock callback function; you prepare and open three cursor specifications, and then establish a lock callback function with a call to `sqlsetlogoption()`. If a lock is encountered while fetching rows for any of the previous three cursor requests, the application waits indefinitely for the lock to be released (the default behavior). However, if you prepare and open a fourth cursor specification and a lock is encountered while fetching rows for this request, your callback function is called, because the fourth cursor request was prepared after you established the callback function.

- When you call `sqlsetlogoption()` with the ESQL_LOCKFUNC option, it establishes your callback function for all subsequent SQL statements prepared or opened during the session. (This callback function has no effect on previous statements prepared or opened.) Setting ESQL_LOCKFUNC implicitly sets *param* to the NULL value. (You can use this effect to reset any parameters you may have specified with prior calls to `sqlsetlogoption()` using the ESQL_LOCKPARM option.) If you set ESQL_LOCKFUNC specifying a NULL value for the `callback` function address (`param`), `sqlsetlogoption()` resets the ESQL resource lock response to the default behavior.

- When you call `sqlsetlogoption()` with the ESQL_LOCKPARM option, it resets your callback function parameter to the current value of `param`. The new value lasts for all subsequent SQL statements prepared or opened during the session until either the callback function designation is changed (using the ESQL_LOCKFUNC option) or you specify a new value (using the ESQL_LOCKPARM option). (This parameter setting has no effect on previous statements prepared or opened.) Setting ESQL_LOCKPARM before your initial setting of ESQL_LOCKFUNC is not an error, but has no effect because `param` is reset to NULL with each setting of ESQL_LOCKFUNC.

EXAMPLE

In the following example, the function `lock_response_USR()` is the `callback` function that `sqlsetlogoption()` registers. It accepts a positive or 0 value, and returns the reciprocal negative or 0 value (the return code) to tell ESQL how to respond to the locked record condition. It prints a message indicating the status of the lock response, where 0 indicates an unexpected record lock and 1 or 2 indicates expected record locks. A 0 causes the application to ignore the unexpected lock and continue execution. A 1 (returned as -1) causes the application to wait for lock grant completion only for a limited time, based on the value of a count (lockcount) that is incremented by the function every time it is called back for this condition. A 2 (returned as -2) reverts to the default behavior and the application waits for lock grant completion indefinitely.

The application example shows `lock_response_USR()` being registered to ignore any unexpected record lock conditions (ESQL_LOCKPARM value is 0). The application can, at another point (not shown), re-register and pass a 1 to `lock_response_USR()` to initiate a lock timeout for the next record lock condition. In this case, ESQL waits for the lock to clear and calls `lock_response_USR()` up to 50 times (++lockcount) before returning an error. At still another point, it can re-register and pass 2 to `lock_response_USR()` to wait for the next record lock condition indefinitely.

The scope of this example is necessarily limited, using ESQL_LOCKPARM to merely pass a reciprocal of the ESQL_LOCKFUNC return condition. A more typical use for ESQL_LOCKPARM might be to pass a pointer to a structure that maintains application context information, including record lock and lock resolution conditions. Then `lock_response_USR()` could manage the value of lockcount, check the latest record lock status, and return the lock handling condition entirely on its own:

```
#include "proesql.h"
static SQLRQHDL *sql0 = (SQLRQHDL *) 0;
static long     SQLCODE;
static int      lockcount = 0;
main(argc,argv)
  int argc;
  char *argv[];
{
               .
               .
               .
  sqlsetlogoption(ESQL_LOCKFUNC, (unsigned long int)lock_reponse_USR);
  sqlsetlogoption(ESQL_LOCKPARM, 0); /* Unexpected lock condition */

  sqlcdbind(&SQLCODE, (char *)0);
  sqldynrq(&sql0, "UPDATE customer SET name = NULL
            WHERE credit-\limit <= 0");
  if (SQLCODE < 0)
  {
    printf("Error executing NULL customer UPDATE\n");
    return;
    }
               .
               .
               .
}
int
lock_response_USR(param, puser, ptty)
  unsigned long int param;
  char          *puser;
  char         *ptty;
{
  if (param == 0)
    printf("User %s at %s has unexpected record lock: Continuing.\n"
             ,puser ,ptty);
  if (param == 1)
    if (++lockcount > 50)
    {
      printf("User %s at %s record lock wait timed out: Error.\n"
               ,puser ,ptty)
      param = 0;
      lockcount = 0;
    }
  if (param != 0 && lockcount <= 50)
    printf("User %s at %s has record locked: Waiting.\n" ,puser ,ptty);
  return(-(param));
}
```

SEE ALSO

sqldynposfetch() — Fetch Row of Dynamic Open Cursor for positioned fetches, sqldynprep() — Prepare a Dynamic SQL Statement for preparing SQL statements

sqlupdrow() — Update Row at Open Cursor

Updates the row at the current position of an open cursor. This function supports the EXEC SQL UPDATE...WHERE CURRENT OF... (positioned UPDATE) statement. If you precompile the statement, the preprocessor generates the r-code file to run it.

This function returns normalized values. For more information, see the function return values in the "General Diagnostics" section in this chapter:

SYNTAX

```
int
sqlupdrow ( sqln, flag, prequest, pvformat, psqlv )
  SQLRQHDL    *sqln;    /* INPUT */
  int         flag;          /* INPUT */
  char        *prequest;     /* INPUT */
  char        *pvformat;    /* INPUT */
  char         *psqlv[];     /* INPUT */
```

sqln

A pointer to the request handle for the open cursor. For more information on request handles, see the "SQL Request Handles" section.

flag

An execution flag that ESQL initializes.

prequest

A pointer to the text of the statement. If you precompile the statement, the text contains the name of the r-code file associated with the statement.

pvformat

A pointer to a format string that describes the types and lengths of the embedded host variables.

psqlv

A pointer to the host variables to which the current SQL request refers.

EXAMPLE

This example updates the row at the position of cursor x (associated with request handle sql0), adding 100 to the value of column max-credit:

```
#include "proesql.h"
static SQLRQHDL *sql0 = (SQLRQHDL *) 0;
            .
            .
            .
    /* EXEC SQL UPDATE CUSTOMER
       SET max\-credit = max\-credit + 100.0
       WHERE CURRENT OF x; */
   {
static char sqltxt[] = {
'U','P','D','A','T','E',',','C','U','S','T','O','M','E','R',',','S','E','T',',','m
',','a','x','-','c','r','e,'d','i','t',',','=',',','m','a','x','-','c','r','e','d,
'i','t',',','+',',','1','0','0','.','0',',','W','H','E','R','E',',','C','U','R','R',
'E','N','T',' ','O','F',',','x','.',0};
static char sqlfmt[] = {0};
       sqlcdbind(&SQLCODE, (char *)0);
       sqlupdrow(sql0,(int)0x0100,sqltxt,sqlfmt, (char **) 0);
       if (SQLCODE < 0) goto err;
    }
```

SEE ALSO

sqldelrow() — Delete Row at Open Cursor for executing positioned DELETE statements, sqldynrq() — Execute a Dynamic Non-SELECT Statement for updating rows of dynamic cursors, sqlrequest() — Execute a Static ESQL Statement for executing non-positioned static ESQL statements.

A

Progress ESQL and ANSI Standards

This appendix lists the Progress/ESQL extensions to and variances from the SQL 1989 ANSI standard X3.135-1989 Level 2 as verified by NIST Test Suite Version 3.0. Quoted phrases denote references directly from the standard, and words between bold angle brackets (**<>**) denote bracketed terms directly from the standard.

This appendix describes:

- Data type conversion rules
- Progress/ESQL extensions not mentioned in the ANSI standard
- ANSI standard behavior available when using the -Q and -Q2 startup parameters
- Progress/ESQL extensions that the ANSI standard explicitly prohibits

NOTES

- Progress/SQL also generally conforms to these extensions and variances plus additional extensions not included in Progress/ESQL.
- The Progress/ESQL C preprocessor does not support the following features of the new ANSI C standard:
  - Trigraphs cannot occur anywhere in the source file.
  - String continuations, indicated by quoted strings separated by white space, may not be used for string literals within the SQL specific sections of a program. They may be used in the Progress 4GL code.

A.1 Data Type Conversion Rules

These rules affect data type conversions and compatibilities during certain SQL operations. For more information on the variety of data type conversions available in ESQL, see Appendix B, "ESQL Data Type Compatibilities."

A.1.1 Assignment Rules

These are the rules for numeric assignments:

- You can assign approximate numeric values to approximate numeric targets (database columns or host language variables).
- You can assign exact numeric values to exact numeric targets; the assignment does not lose any leading significant digits.
- You can assign approximate numeric values to exact numeric targets, with possible loss of precision.

  NOTE: Progress/ESQL extension — The 1989 ANSI standard does not allow you to assign approximate numeric values to exact numeric targets.

The rules for character values depend on whether the target is a table column or a host language variable:

- FETCH, SELECT INTO — You can assign character values to character targets; indicators signal truncation.
- INSERT — You can insert character strings into character targets; the length of the target must be greater than or equal to the length of the source.
- UPDATE — Character strings can be updated targets that are shorter than the source.

  NOTE: Progress/ESQL extension — The 1989 ANSI standard requires target length to be greater than or equal to the source.

A.1.2 Comparison Rules

The ANSI SQL comparison rules are generally defined. The only incompatibility is case sensitivity: string comparisons are not case sensitive unless you set the -Q switch (see the "Standard Behavior Available Only with -Q or -Q2" section).

A.1.3 UNION Data Type Compatibility

There is one rule for how you can combine table columns from UNION queries: data types of corresponding columns must be assignment compatible.

NOTE: Progress/ESQL extension — The 1989 ANSI standard states that the data types of corresponding columns must be identical.

A.2 Progress/ESQL Not Addressed in 1989 SQL Standard

The following Progress/ESQL extensions are not included in the standard:

- Dynamic ESQL
- DATE data type
- LOGICAL data type
- DECLARE CURSOR . . . FOR READ ONLY
- DECLARE CURSOR . . . FOR UPDATE
- DROP [TABLE | INDEX | VIEW]
- ALTER TABLE
- CREATE INDEX
- Column attributes (FORMAT, LABEL, and COLUMN-LABEL)
- BEGINS predicate
- Progress functions, such as RECID, CAN-DO, and CONTAINS
- Subscripted fields, and vector (array) fields
- + string concatenation operator
- Names more than 18 characters long
- OUTER JOIN support

A.3 Standard Behavior Available Only with -Q or -Q2

The following ANSI standard features are available only when you use the specified startup parameter:

- CREATE TABLE — Columns are not case sensitive unless you set -Q or -Q2.
- UPDATE — Interim uniqueness conflict avoided when you set -Q or -Q2.
- Decimal precision — Set to 15 when you set -Q or -Q2.
- Character comparisons — Not case sensitive unless you set -Q or -Q2.
- Blank padding — Destination fields (host language variables) are blank padded when you set -Q or -Q2.
- Guaranteed serializability — Uses Progress record locking rules unless you set -Q2.
- Division by zero — Division by zero returns the UNKNOWN value unless you set -Q, which causes an error to be raised.

A.4 Progress/ESQL Extensions to 1989 SQL

The standard explicitly forbids the following Progress/ESQL extensions:

- GROUP BY <scalar expression> — SQL 89 allows only a column reference.
- GROUP BY or HAVING in <subquery>.

  NOTE: SQL 89 prohibits this if the subquery is part of a comparison predicate; Progress allows this.

- ORDER BY column not in <select list>.

  NOTE: Progress/ESQL allows this if the query does not contain a clause (GROUP BY, DISTINCT, UNION, etc.) that causes the cursor to use a temporary result table.

- ORDER BY <scalar expression> — SQL 89 allows only a column reference.
- ORDER BY and cursors — SQL 89 ORDER BY cursors are read only.

  NOTE: Progress/ESQL cursors with ORDER BY are updatable unless the cursor uses a temporary result table.

- ORDER BY in <subquery>.

- ESCAPE character in LIKE predicate — SQL 89 allows only a single-character literal.

  NOTE: Although multiple-character string literals are allowed, only the first character is used.

- Table name repeated in FROM clause.

  NOTE: You may repeat a table name in a FROM clause without a correlation name. The duplicate is ignored.

- DISTINCT specified more than once in a subquery.
- Updatable query expression with a subquery.
- DISTINCT set function as an operand of a dyadic operator.
- INSERT . . . SELECT specifies the same table for INTO as SELECT.
- DELETE FROM <table> WHERE specifies <table> in a subquery.
- UPDATE specifies same column more than once in SET list.
- SELECT INTO with GROUP BY.
- SELECT INTO with HAVING.

B

ESQL Data Type Compatibilities

When writing an ESQL application, there are several data type compatibility issues among Progress, SQL, static and Dynamic ESQL, and C. This appendix describes these data type correspondences and how they apply to an ESQL application.

Progress static and Dynamic ESQL offer different degrees of flexibility to exchange data between SQL table columns and C variables of different data types. Using static ESQL, you must declare C variables using a specific range of data types that match the Progress data type of the corresponding table column. Using Dynamic ESQL, Progress performs automatic data conversion between a much broader range of C and Progress data types.

This appendix describes:

- The relationship between SQL and core Progress data types
- Standard C type definitions used in both static and Dynamic ESQL
- Data type matching between SQL and C data types in static ESQL
- Data type matching between SQL and C data types in Dynamic ESQL, and Dynamic ESQL output/input data conversion

B.1 SQL and Core Progress Data Types

When you execute a CREATE TABLE or ALTER TABLE statement in ESQL, you can specify supported SQL standard data types for the columns. However, Progress defines these columns using five core storage data types. Table B–1 lists the relationship between these types and their related storage capabilities.

Table B–1: SQL and Progress Core Data Types

SQL Data Type	Progress Data Type	Storage Capacity
CHARACTER [(*n*)]	CHARACTER	Variable-length character strings, where *n* specifies the number of characters in the default display format, "*x*(*n*)". Record size constrains the maximum length.
DATE[1]	DATE	Date values between 1/1/32768 B.C. and 12/31/32767 A.D.
DECIMAL [(*m* [,*n*])] DOUBLE PRECISION FLOAT [(*m*)] NUMERIC [(*m* [,*n*])] REAL	DECIMAL	Decimal numbers up to 50 digits long, with a maximum of 10 digits to the right of the decimal point,[2] where *m* specifies the number of significant digits and *n* specifies the number of digits to the right of the decimal point.
INTEGER	INTEGER	Whole numbers in the range -2,147,483,648 to 2,147,483,647.
LOGICAL[1]	LOGICAL	Boolean values such as YES/NO or TRUE/FALSE.

1 These are non-standard, Progress-defined SQL data types.

2 A maximum of 15 digits to the right of the decimal point is allowed when running with the ANSI SQL (-Q) or ANSI SQL Client (-Q2) startup parameter.

B.2 ESQL Standard C Data Type Definitions

The `proesql.h` header file in your ESQL installation directory contains several data type definitions to support ANSI standard data transfers between SQL tables and your ESQL application:

```
typedef char sqldate[13];
typedef char sqlbool;
typedef char sqldec[53];
```

Because they are essentially character string data types, the sqldate and sqldec type definitions reserve an extra byte for a null terminator. These type definitions are useful for providing compatibility between certain SQL and C data types, as shown in the following sections.

You can also use many standard C data types as described in this appendix. However, the int data type is non-standard for ESQL. If you use this data type to declare host language variables, the preprocessor produces a warning. For more information on using the standard C data types to define host language variables, see Chapter 2, "Programming with ESQL."

B.3 Static ESQL Data Type Matching

In static ESQL, a host variable must have a data type that is compatible with the SQL table column. Data conversion is not supported in static ESQL. For information on data conversion in dynamic ESQL, see the "Dynamic ESQL Data Type Matching and Conversion" section. Table B–2 shows the compatible C language data types for each SQL data type.

Table B–2: Progress/SQL Data Types and C Equivalents *(1 of 2)*

SQL Data Types	C Data Types
CHARACTER	char
DATE[1]	sqldate (char)
DECIMAL, NUMERIC	sqldec (char)
FLOAT, REAL	float[2]
DOUBLE PRECISION	double[2]
INTEGER	long (int)[3]

Table B–2: Progress/SQL Data Types and C Equivalents *(2 of 2)*

SQL Data Types	C Data Types
SMALLINT	short
LOGICAL[1]	sqlbool (char)

[1] These are non-standard, Progress-defined SQL data types.

[2] You can move data from the corresponding SQL type to these data types, but with a possible loss of precision.

[3] The `int` C data type is non-standard for ESQL.

NOTE: RAW data types are not supported in Progress/ESQL. An attempt to retrieve data of this type will result in an error.

Any numeric SQL data type (DECIMAL, FLOAT, INTEGER, NUMERIC, REAL, and SMALLINT) is compatible with any of the numeric C data types listed in Table B–2.

Each SQL data type maps to a core Progress data type in the database. For example, an SQL column specified as REAL is stored as a Progress DECIMAL value. For more information on how SQL data types map to Progress data types, see the "SQL and Core Progress Data Types" section.

B.4 Dynamic ESQL Data Type Matching and Conversion

Dynamic ESQL supports the following two methods of data exchange between SQL tables and C, depending on the operation.

- Host variables to store dynamic column fetches and specify dynamic parameter marker values for several dynamic SQL statements where the exact syntax is known ahead of time.
- An SQL descriptor area for all other data exchange statements.

For the dynamic EXEC SQL FETCH...INTO... statement, for example, ESQL performs data conversion on host variables using the same criteria supported for exchanges using an SQL descriptor area.

B.4.1 The SQL Descriptor Area and Data Type Conversion

You declare an SQL descriptor area using the SQLDA_T and SQLVAR_T structures defined in the proesql.h header file. The SQLVAR_T structure describes the C data type and storage characteristic for one column of an SQL table. The following tables describe the members of this structure that specify the data type of the column.

The TYPE Member

Table B–3 shows the supported relationship between each SQL data type, its corresponding SQLVAR_T->TYPE code (defined in proesql.h), and the required C data type for storage allocated for the SQLVAR_T->TYPE.

Table B–3: Matching Dynamic SQL and C Data Types

SQL Data Type	SQLVAR_T->TYPE	C Data Type
CHARACTER	SQLDACHAR	char
DATE[1]	SQLDADATE	sqldate (char)
DECIMAL	SQLDADEC	sqldec (char)
DOUBLE PRECISION	SQLDADBL	double
FLOAT	SQLDAFLT	float
INTEGER	SQLDAINT	long
LOGICAL[1]	SQLDALOG	sqlbool (char)
NUMERIC	SQLDANUM	sqldec (char)
REAL	SQLDAREAL	float
SMALLINT	SQLDASML	short

[1] These are non-standard, Progress-defined SQL data types.

If you use the EXEC SQL DESCRIBE... statement or the `sqldyndesc()` function, ESQL-LIB fills in the SQL descriptor area (SQLVAR_T->TYPE) with appropriate data types for the statement columns. If you prepare the SQL descriptor for dynamic parameter markers using the `sqldyndescinp()` function, ESQL-LIB fills in the SQL descriptor area with the data types of the dynamic parameter markers.

If you fill in the descriptor area, before a FETCH, EXECUTE, or OPEN, the actual Progress data type of the table column might not correspond to the SQL data type that you specify in the SQL descriptor area. If possible, Progress converts the data into a meaningful form using the available source and target data types, as described in the "Output Data Conversion" and "Input Data Conversion" sections.

The PRECISION, SCALE, and LENGTH Members

As part of data conversion, ESQL uses the PRECISION, SCALE, and LENGTH members in the SQLVAR_T structure depending on the Progress data type involved in the conversion.

Table B–4 shows the actual use of PRECISION for each Progress data type after a call to `sqldyndesc()`.

Table B–4: Precision of SQL Descriptor Data Types

Progress Data Type	SQLVAR_T->PRECISION
CHARACTER	N/A
DATE	N/A
DECIMAL	The defined maximum number of digits. For example, the precision of a column defined as `DECIMAL(10,3)` is 10.
INTEGER	N/A
LOGICAL	N/A

The SCALE member specifies the maximum number of digits to the right of the decimal point for a precisely scaled numeric data type.

Table B–5 shows the actual use of SCALE for each Progress data type after a call to `sqlsqldyndesc()`.

Table B–5: Scale of SQL Descriptor Data Types *(1 of 2)*

Progress Data Type	SQLVAR_T->SCALE
CHARACTER	N/A
DATE	N/A

Table B–5: Scale of SQL Descriptor Data Types *(2 of 2)*

Progress Data Type	SQLVAR_T->SCALE
DECIMAL	The defined number of digits to the right of the decimal place. For example, the scale of a column defined as DECIMAL(10,3) is 3. For approximate floating point data types in the SQLVAR_T structure (for example, SQLDAFLT or SQLDADBL), the scale is undefined because the number of digits to the right of the decimal point is not fixed.
INTEGER	N/A
LOGICAL	N/A

The LENGTH member specifies the maximum number of bytes returned to the application when data is transferred to its default C data type.

Table B–6 shows the actual usage of LENGTH for each Progress data type after a call to sqldyndesc().

Table B–6: Output Length of SQL Descriptor Data Types

Progress Data Type	SQLVAR_T->LENGTH
CHARACTER	The defined length of the column, not including the null termination byte. For example, the output length of a column defined as CHAR(10) is 10, with one extra byte for the null terminator.
DATE	N/A
DECIMAL	N/A
INTEGER	N/A
LOGICAL	N/A

B.4.2 Output Data Conversion

The following tables describe the conversion results when fetching data from an SQL column to an SQL descriptor area. Each table describes the data conversion for a column according to its core Progress data type.

All column data converted to character data types (for example, SQLDACHAR and SQLDATE) is null-terminated regardless of whether this truncates the result.

Table B–7 describes data conversion from a CHARACTER column to all supported SQL descriptor data types.

Table B–7: Dynamic Output from a CHARACTER Column *(1 of 2)*

Converted to . . .	Value Output	Indicator Variable	Error Conditions
SQLDACHAR	As much of the character string value as fits in the available storage space.	If the available storage space is too small, set to the number of bytes required for the complete Column value.	None.
SQLDADATE	A valid SQL date string.	N/A	Invalid SQL date string in Column.
SQLDADEC SQLDANUM	As much of the valid SQL decimal number string as fits in the available storage space, with all leading and trailing white space removed.	If the available storage space is too small, set to the number of bytes required for the complete Column value.	Invalid decimal string in Column. Decimal string followed by non-white-space characters. Decimal string with digit precision > SQLVAR_T->PRECISION.
SQLDADBL SQLDAFLT SQLDAREAL	A valid SQL decimal number.	N/A	Invalid decimal string in Column. Decimal string followed by non-white-space characters. Decimal string value is too large for target (SQLDADBL or SQLDAFLT) data type.

Table B–7: Dynamic Output from a CHARACTER Column *(2 of 2)*

Converted to . . .	Value Output	Indicator Variable	Error Conditions
SQLDAINT	A valid long integer or long integer truncated from a decimal number string in Column.	N/A	Invalid decimal string in Column. Decimal string followed by non-white-space characters. String numeric value of Column > 2147483647 or < -2147483648.
SQLDALOG	0 or 1.	N/A	Column does not contain a valid integer or decimal string value equivalent to 0 or 1 (for example, “1.000" or “0001"). Decimal number followed by non-white-space characters.
SQLDASML	A valid short integer or short integer truncated from a decimal number string in Column.	N/A	Invalid decimal string in Column. Decimal string followed by non-white-space characters. String numeric value of Column > 32767 or < -32768.

Table B–8 describes data conversion from an INTEGER column to all supported SQL descriptor data types.

Table B–8: Dynamic Output from an INTEGER Column

Converted to . . .	Value Output	Indicator Variable	Error Conditions
SQLDACHAR	A valid integer number string.	N/A	Storage space not large enough for integer string value.
SQLDADATE	Not supported.	N/A	Output to SQLDADATE attempted.
SQLDADEC SQLDANUM	A valid decimal string with a numeric value equivalent to the integer value in Column.	N/A	Storage space not large enough for decimal string value.
SQLDADBL SQLDAFLT SQLDAREAL	A floating-point value equivalent to the integer value in Column.	N/A	None.
SQLDAINT	The integer value of Column.	N/A	None.
SQLDALOG	0 or 1.	N/A	Column does not contain a value of 0 or 1.
SQLDASML	The short integer value of Column.	N/A	Value of Column > 32767 or < -32768.

Table B–9 describes data conversion from a DECIMAL column to all supported SQL descriptor data types.

Table B–9: Dynamic Output from a DECIMAL Column *(1 of 2)*

Converted to . . .	Value Output	Indicator Variable	Error Conditions
SQLDACHAR	A valid decimal number string.	If the available storage space is too small to store all decimal digits of the Column value, set to the number of bytes required to store the complete value of Column.	Storage space not large enough for all significant digits to the left of the decimal point.
SQLDADATE	Not supported.	N/A	Output to SQLDADATE attempted.
SQLDADEC SQLDANUM	A valid decimal number string.	If the available storage space is too small to store all decimal digits of the Column value, set to the number of bytes required to store the complete value of Column.	Storage space not large enough for all significant digits to the left of the decimal point.
SQLDADBL SQLDAFLT SQLDAREAL	A floating-point value equivalent to or approximately equivalent to the decimal value in Column.	N/A	Value is greater than maximum floating-point value (for SQLDAFLT type).
SQLDAINT	The decimal-truncated integer value of Column.	N/A	Decimal value of Column > 2147483647 or < -2147483648.

Table B–9: Dynamic Output from a DECIMAL Column *(2 of 2)*

Converted to . . .	Value Output	Indicator Variable	Error Conditions
SQLDALOG	0 or 1.	N/A	Column does not contain a value of 0 or 1.
SQLDASML	The decimal-truncated short integer value of Column.	N/A	Decimal value of Column > 32767 or < -32768.

Table B–10 describes data conversion from a DATE column to all supported SQL descriptor data types.

Table B–10: Dynamic Output from a DATE Column *(1 of 2)*

Converted to . . .	Value Output	Indicator Variable	Error Conditions
SQLDACHAR	A string representation of the valid SQL date in Column.	If the available storage space is too small to store the entire date, set to the number of bytes required to store the complete date value of Column.	Column contains an invalid SQL date.
SQLDADATE	A string representation of the valid SQL date in Column.	N/A	Column contains an invalid SQL date.
SQLDADEC SQLDANUM	Not supported.	N/A	Output to SQLDADEC or SQLDANUM attempted.

Table B–10: Dynamic Output from a DATE Column *(2 of 2)*

Converted to . . .	Value Output	Indicator Variable	Error Conditions
SQLDADBL SQLDAFLT SQLDAREAL	Not supported.	N/A	Output to SQLDADBL, SQLDAFLT, or SQLDAREAL attempted.
SQLDAINT	Not supported.	N/A	Output to SQLDAINT attempted.
SQLDALOG	Not supported.	N/A	Output to SQLDALOG attempted.
SQLDASML	Not supported.	N/A	Output to SQLDASML attempted.

Table B–11 describes data conversion from a LOGICAL column to all supported SQL descriptor data types.

Table B–11: Dynamic Output from a LOGICAL Column *(1 of 2)*

Converted to . . .	Value Output	Indicator Variable	Error Conditions
SQLDACHAR	0 or 1.	N/A	SQLVAR_T-> LENGTH is set to 0.
SQLDADATE	Not supported.	N/A	Output to SQLDADATE attempted.
SQLDADEC SQLDANUM	0 or 1.	N/A	None.

Table B–11: Dynamic Output from a LOGICAL Column *(2 of 2)*

Converted to . . .	Value Output	Indicator Variable	Error Conditions
SQLDADBL SQLDAFLT SQLDAREAL	0 or 1.	N/A	None.
SQLDAINT	0 or 1.	N/A	None.
SQLDALOG	0 or 1.	N/A	None.
SQLDASML	0 or 1.	N/A	None.

B.4.3 Input Data Conversion

The following tables describe the conversion results when updating or inserting an SQL row from data in an SQL descriptor area. These conversion rules also apply in all cases when providing values for Dynamic parameter markers from an SQL descriptor area. Each table describes the data conversion for one SQL descriptor data type or class of data types.

Conversion from SQLDACHAR

Table B–12 describes data conversion from an SQL character (SQLDACHAR) data type to all supported core Progress column data types.

Table B–12: Dynamic Input from SQL Character Data Types *(1 of 2)*

Converted to . . .	Value Input	Error Conditions
CHARACTER	The value of the SQL character data.	None.
DATE	The date value of the SQL character data.	SQL character data contains an invalid SQL date string.

Table B–12: Dynamic Input from SQL Character Data Types *(2 of 2)*

Converted to . . .	Value Input	Error Conditions
DECIMAL	The decimal value of the string numeric in the SQL character data.	SQL character data contains an invalid decimal string value. Precision of the character string numeric > 50. Number of digits to the right of the decimal point > 15.
INTEGER	The integer value of the string numeric in the SQL character data, decimal truncated if necessary.	SQL character data contains an invalid decimal string value. The truncated decimal value is > 2147483647 or < -2147483648.
LOGICAL	TRUE (1) or FALSE (0).	SQL character data does not contain a valid integer or decimal string value equivalent to 0 or 1.

Conversion from SQLDADATE

Table B–13 describes data conversion from an SQL date (SQLDADATE) data type to all supported core Progress column data types.

Table B–13: Dynamic Input from SQL Date Data Types *(1 of 2)*

Converted to . . .	Value Input	Error Conditions
CHARACTER	The string-formatted value of the SQL date data.	None.
DATE	The Progress date value of the SQL date data.	SQL date data contains an invalid SQL date string.
DECIMAL	Not supported.	Input from SQL date data attempted.

Table B–13: Dynamic Input from SQL Date Data Types *(2 of 2)*

Converted to . . .	Value Input	Error Conditions
INTEGER	Not supported.	Input from SQL date data attempted.
LOGICAL	Not supported.	Input from SQL date data attempted.

Conversion from SQLDADEC or SQLDANUM

Table B–14 describes data conversion from an SQL decimal (SQLDADEC or SQLDANUM) data type to all supported core Progress column data types.

Table B–14: Dynamic Input from SQL Decimal Data Types

Converted to . . .	Value Input	Error Conditions
CHARACTER	The value of the SQL decimal data.	None.
DATE	Not supported.	Input from SQL decimal data attempted.
DECIMAL	The decimal value of the string numeric in the SQL decimal data.	SQL decimal data contains an invalid decimal string value. Precision of the SQL decimal string numeric > 50. Number of digits to the right of the decimal point > 15.
INTEGER	The integer value of the string numeric in the SQL decimal data, decimal truncated if necessary.	SQL decimal data contains an invalid decimal string value. The truncated decimal value is > 2147483647 or < -2147483648.
LOGICAL	TRUE (1) or FALSE (0).	SQL decimal data does not contain a valid integer or decimal string value equivalent to 0 or 1.

Conversion from SQLDAFLT, SQLDADBL, or SQLDAREAL

Table B–15 describes data conversion from an SQL floating-point (SQLDAFLT, SQLDADBL, or SQLDAREAL) data type to all supported core Progress column data types.

Table B–15: Dynamic Input from SQL Floating Point Data Types

Converted to . . .	Value Input	Error Conditions
CHARACTER	The string-formatted value of the SQL floating-point data.	None.
DATE	Not supported.	Input from SQL floating-point data attempted.
DECIMAL	The decimal value of the SQL floating-point data.	None.
INTEGER	The decimal-truncated value of the SQL floating-point data.	The truncated decimal value is > 2147483647 or < -2147483648.
LOGICAL	TRUE (1) or FALSE (0).	SQL floating-point data contains a value other than 0 or 1.

Conversion from SQLDAINT or SQLDASML

Table B–16 describes data conversion from an SQL integer (SQLDAINT or SQLDASML) data type to all supported core Progress column data types.

Table B–16: Dynamic Input from SQL Integer Data Types *(1 of 2)*

Converted to . . .	Value Input	Error Conditions
CHARACTER	The string-formatted value of the SQL integer data.	None.
DATE	Not supported.	Input from SQL integer data attempted.
DECIMAL	The decimal value of the SQL integer data.	None.

Table B–16: Dynamic Input from SQL Integer Data Types *(2 of 2)*

Converted to . . .	Value Input	Error Conditions
INTEGER	The value of the SQL integer data.	None.
LOGICAL	TRUE (1) or FALSE (0).	SQL integer data contains a value other than 0 or 1.

Conversion from SQLDALOG

Table B–17 describes data conversion from an SQL logical (SQLDALOG) data type to all supported core Progress column data types.

Table B–17: Dynamic Input from SQL Logical Data Types

Converted to . . .	Value Input	Error Conditions
CHARACTER	0 or 1.	None.
DATE	Not supported.	Input from SQL logical data attempted.
DECIMAL	0 or 1.	None.
INTEGER	0 or 1.	None.
LOGICAL	TRUE (non-zero) or FALSE (0).	None.

C

V6 HLI to V9 ESQL Migration

This appendix describes the changes you make to migrate an existing C application from Progress Version 6 Host Language Interface (HLI) to Versions 9 ESQL. These changes depend on how your application builds SQL statements. Thus, the following sections describe how to migrate your application if it uses:

- The SQL processor to build SQL statements
- ESQL-LIB (direct function calls) to build SQL statements

C.1 Migrating SQL Preprocessor-based Applications

This section describes how to change your application if it uses the SQL processor to make SQL calls. If your application also uses ESQL-LIB to make some SQL calls, also see the "Migrating ESQL-LIB-based Applications" section.

C.1.1 Using Dynamic ESQL

If you are using dynamic ESQL, add the SQL keyword to the DESCRIBE statement. Thus, this Version 6 statement:

```
EXEC SQL DESCRIBE DEMOSTMT USING DESCRIPTOR psqlda;
```

becomes this Version 9 statement:

```
EXEC SQL DESCRIBE DEMOSTMT USING SQL DESCRIPTOR psqlda;
```

Change the USING clause in the FETCH statement to INTO SQL. Thus, this Version 6 statement:

```
EXEC SQL FETCH C1 USING DESCRIPTOR psqlda;
```

becomes this Version 9 statement:

```
EXEC SQL FETCH C1 INTO SQL DESCRIPTOR psqlda;
```

C.1.2 Calling sqllogout()

Unlike Version 6, a Version 9 call to `sqllogout()` does not exit your application, but returns control to your application. Carefully check all of your calls to `sqllogout()`. If your application expects to exit at that point, call `exit()` following the call to `sqllogout()`. Note also that in Version 9, the `exit()` function is no longer redefined to call `sqllogout()`; `exit()` now has its standard C run-time library behavior.

C.1.3 Using Host Language Variables

Using the C `int` data type for host language variables in either static or dynamic ESQL is nonstandard. Therefore, the processor flags any instances of its use with a warning. Use either short or long.

C.1.4 Preparing the Executable

Before you run the SQL processor and recompile your application, note that the name of `sqlhli.h` in Version 6 is changed to `proesql.h`, and the installation path for `proesql.h` and `sqlcpp` (the processor) is changed to `$DLC/probuild/esqlc`.

Before you relink the executable, generate a new load script with the PROBUILD utility. The load script name and contents have changed.

C.2 Migrating ESQL-LIB-based Applications

The following sections describe how to change your application if it uses ESQL-LIB to make SQL calls. If your application also uses the SQL processor, also see the "Migrating SQL Preprocessor-based Applications" section.

C.2.1 Changing Calls to sqlcdbind()

In Versions 9, you must add an additional parameter to `sqlcdbind()` for the value of the SQLSTATE variable. For more information, see the description of SQLSTATE in Chapter 2, "Programming with ESQL" and the reference entry for `sqlcdbind()` in Chapter 4, "ESQL-LIB Reference."

C.2.2 Changing sqlv[] Array References

In Version 6, the `sqlv[]` array is defined as a global static variable. In Versions 9, this array is dynamically allocated and passed as a parameter to the following functions:

- `sqlfetch()` — static FETCH
- `sqlupdrow()` — static positioned UPDATE
- `sqlrequest()` — static EXECUTE

C.2.3 Changing Function Return Behavior

In Version 6, an HLI function can image-exit under certain conditions (for example, if a fatal Progress error occurs). In Version 9, all ESQL-LIB functions return control to the application, regardless of conditions. If a fatal Progress error occurs, the application recognizes this from an SQLCODE value of -99 and can take appropriate action on its own. In this case, further calls to all ESQL-LIB functions, except `sqlgetmsg()`, have no effect.

Also unlike Version 6, a Version 9 call to `sqllogout()` does not exit, but returns control to your application. Carefully check all of your calls to `sqllogout()`. If your application expects to exit at that point, `exit()` following the call to `sqllogout()`. Note also that in Version 7 and Version 8, the `exit()` function is no longer redefined to call `sqllogout()`; `exit()` now has its standard C run-time library behavior.

C.2.4 Replacing Calls to sqldynopen()

The Version 6 dynamic ESQL `sqldynopen()` function no longer exists. You must replace it with calls to these dynamic ESQL functions:

- `sqldynprep()` — To prepare the SELECT statement
- `sqldynsetn()` — To associate a cursor name with the query
- `sqldyndesc()` — To retrieve column attribute descriptions
- `sqldynexec()` — To execute the statement

C.2.5 Replacing Calls to sqlclose()

The Version 6 `sqlclose()` function no longer exists in Version 9. You must replace it with a call to `sqldynclose()`.

C.2.6 Removing Obsolete Functions

In Version 9, the following functions no longer exist, and you must remove them from your application:

- `sqlrtnmem()`
- `sqlprepd()`
- `sqlnprepd()`

C.2.7 Preparing the Executable

Before compiling your application, note that the name and installation path of `sqlhli.h` have changed. It is now named `proesql.h` and is installed in `$DLC/probuild/esqlc`.

Index

A

-a Preprocessor parameter 3–11

ALLOCATE DESCRIPTOR
  statement 2–79
    cursor specifications 2–67
    using for parameter markers 2–75

ANSI SQL (-Q) startup parameter 3–19

ANSI SQL Client (-Q2) startup parameter 3–19

ANSI standards 1–1
  compliance and variance A–1
    comparison A–2
    data type conversion A–2
    forbidden extensions A–4
    standard with -Q or -Q2 only A–4
    unaddressed extensions A–3
    UNION data types A–3
  flagging code deviations 3–11

Application names 2–37

-applnam Preprocessor parameter 3–11

Architecture of ESQL/C 1–5
  configuration and deployment 1–7
  large client model 1–5
  small client model 1–5
  small clients compared to large clients 1–6

Array references
  parameter markers 2–72
  SQLVAR_T structures 2–56

Audience xi

Auto-starting local OIB/OID 3–26

B

BEGIN DECLARE SECTION
  statement 2–23

Bold typeface
  as typographical convention xii

Building executables
  ESQL instructions 3–6
  ESQL overview 1–9

C

C language data types
  for host language variables 2–23

Callback lock function 2–46

CAN-DO function 2–41
  retrieving value 2–42
  WHERE clause 2–44

CHARACTER data type
  blank-padding output 3–19
  Progress/SQL 2–15

Choosing ESQL/C configurations 3–2

clidemo.c file 1–8
  compiling and running 3–27

CLOSE statement, Dynamic 2–86
  cursor specifications 2–69
  using with parameter markers 2–76

COLCNT member of SQLDA_T 2–61

COMMIT WORK statement 2–41

Compiling ESQL/C source files 3–12
  MS-Windows 3–13
  UNIX systems 3–12

Compiling preprocessed source files 1–14

Conditional actions 2–18

Configuration 1–7
  choosing a model 3–2

Connecting databases
  multiple 2–38
  SQL Preprocessor 3–8
  sqlconn() compared to sqllogin() 2–37
  startup parameters 2–38
  using sqlconn() 2–38
  using sqllogin() 2–34

Connecting small client and OID 3–21

Connection parameters 2–38

Constants
  Static ESQL 4–3

Conversion
  data types
    Dynamic ESQL B–4
    Static ESQL B–3

Correlation names 2–39

CREATE SCHEMA statement 2–20

CTRL-C interrupts 2–46

Cursor specifications
  Dynamic 2–65
    executing 2–66
    parameter markers 2–55
    See also Non-cursor specifications requests
  Static 2–65
    example 2–4

Cursors
  Dynamic
    names 2–81
  Static
    SQL request handles 4–2

D

DATA member of SQLVAR_T 2–62

Data type matching 2–17

Data types
  C language 2–23, B–3
  CHARACTER type 2–15
  conversion
    Dynamic ESQL B–4
    Dynamic input B–14
    Dynamic output B–8
    Static ESQL B–3
  DATE type 2–17
  DECIMAL type 2–16, 2–17

DOUBLE PRECISION type 2–16
Dynamic conversion
CHARACTER output B–8
DATE output B–12
DECIMAL output B–11
INTEGER output B–10
LOGICAL output B–13
SQLDACHAR input B–14
SQLDADATE input B–15
SQLDADBL input B–17
SQLDADEC input B–16
SQLDAFLT input B–17
SQLDAINT input B–17
SQLDALOG input B–18
SQLDANUM input B–16
SQLDAREAL input B–17
SQLDASML input B–17
FLOAT type 2–16
INTEGER type 2–15
LOGICAL type 2–17
matching in Dynamic ESQL B–4
matching in Static ESQL B–3
parameter marker 2–72
Progress/ESQL 2–15
REAL type 2–17
SMALLINT type 2–16
SQL compared to core Progress B–2
sqlbool 2–22
sqldate 2–22
sqldec 2–22

Database managers
defined 1–16

Databases
connection 2–37
parameter files 3–26
connection parameters 2–38
correlation names 2–39
multiple connection 2–38
name qualifiers 2–39
number connected 2–38
schema trigger access 2–13
See also Schema triggers
schema user names 2–20
small client connections 3–25

DataServers
using ESQL/C 3–6

DATE data type
Progress/SQL 2–17

DDL statements
defined 2–2
precompiling 3–8
prepared execution 2–79
Progress/ESQL 2–20
transactions 2–41

DEALLOCATE DESCRIPTOR statement
Dynamic 2–86

-debug Preprocessor parameter 3–10

DECIMAL data type
increasing scale maximum 3–19
Progress/SQL 2–16, 2–17

DECLARE CURSOR statement
Dynamic 2–80
cursor specifications 2–54, 2–68
using with parameter markers 2–75

Declaring
host language variables 2–23

Demo applications. *See also* ESQL/C demo applications

DESCRIBE statement 2–80
cursor specifications 2–54, 2–68

Designing and programming 1–10

Diagnostics
ESQL-LIB 4–7

Directory
ESQL installation 1–7

Disambiguating names 2–39

Disconnecting databases

sqldiscon() 2–39
transactions 2–40, 4–17, 4–24

DML statements
defined 2–2

DOUBLE PRECISION data type
Progress/SQL 2–16

Dynamic Data Exchange
OID direct startup 3–24

Dynamic data type conversion B–4

Dynamic Embedded SQL
compared to Static 2–47
compared to Static ESQL 2–2
data type conversion B–4
input to parameter markers B–14
output from columns B–8
SQL descriptor area B–5
defined 1–2
descriptor areas. *See also* SQL descriptor area
executing cursor specifications. *See also* Prepared execution
executing requests 2–48
execution 2–49
immediate execution
defined 2–49
parameter markers 2–70
prepared execution
defined 2–49
prepared execution. *See also* Prepared execution defined
request string 2–47
requests 2–47
singleton SELECT 2–48
specifying requests 2–47
statements for prepared execution 2–78
using 2–47

Dynamic ESQL
EXECUTE IMMEDIATE statement 2–51
See also Dynamic Embedded SQL

Dynamic parameter markers
cursor specifications 2–55
data type assumptions 2–72
defined 1–2
host language variables 2–76
non-cursor specification requests 2–55
restrictions 2–72
using 2–70
using in applications 2–73
using with ESQL-LIB 2–76, 2–78
using with Progress/ESQL 2–74, 2–76

dyndemo.cc file 1–8
building and running 3–29

E

Embedded SQL for C
demo applications
See also ESQL/C demo applications

Embedded SQL for C (ESQL/C)
ANSI standards 1–1
capabilities 1–2
defined 1–1
dynamic 2–47
managing transactions 2–40
programming 2–1

END DECLARE SECTION statement 2–23

Ending ESQL sessions 2–37

Error handling 2–26
ESQL 2–18
ESQL-LIB 4–7
message logging 3–18
Preprocessor 3–9

Error messages
displaying descriptions xvii

Errors
function return codes 2–33

Progress messages 2–26, 2–33
return codes 2–26
SQLCODE 2–26
SQLSTATE 2–26

ESQL declaration section 2–23

ESQL Log (-sqllog) startup parameter 3–18

ESQL No Padding (-esqlnopad) startup parameter 3–18

ESQL sessions
application names 2–37
database
connection 2–37, 2–38
multiple connection 2–38
name qualifiers 2–39
database disconnection 2–39
transactions 2–40, 4–17, 4–24
ending 2–37
large client CTRL-C interrupts 2–46
lock management 2–44
run-time privileges 2–41
specifying startup parameters 2–35
large clients 2–35
small clients 2–35
techniques 2–35
starting 2–34
transaction management 2–40
committing work 2–41
logging out 2–40
rolling back work 2–41

ESQL/C applications
default lock behavior 2–46
ESQL-LIB function returns 4–9
preparing to run 3–17
running 3–17

ESQL/C demo applications 3–27
Dynamic ESQL 3–29
parameter markers 3–27
Static ESQL 3–27

ESQL/C source files
compiling 3–12

esqldemo.cc file 1–7
building and running 3–27

ESQL-LIB
compared to Progress/ESQL 2–2
defined 1–1
diagnostics 4–7
direct use
conditions allowing 2–1
function returns 4–9
exceptional 4–8
normalized 4–7
functions 4–4
interface defined 1–15
overview 4–2
parameter markers 2–76
reference entries 4–9
SQL descriptor area 4–3
SQL Preprocessor 3–8
SQLCODE variable 4–2
SQLSTATE variable 4–2
V6 to V7 migration C–3

-esqllog startup parameter 3–18

-esqlnopad Preprocessor parameter 3–11

-esqlnopad startup parameter 3–18

Example application
preprocessed ESQL 2–6
unpreprocessed ESQL 2–4

EXCLUSIVE-LOCK option 2–44
SQL requests 2–45

EXEC SQL statement prefixes 2–9

Executables for ESQL/C
functional layers 1–15
linking 3–14
preparing to run 3–17
running 3–17
types 1–14

EXECUTE IMMEDIATE statement 2–51

EXECUTE statement 2–82

Dynamic
using with parameter markers
2–75
prepared execution 2–53

F

FETCH statement
dynamic 2–84
cursor specifications 2–54,
2–69

Files
clidemo.c 1–8
dyndemo.cc 1–8
esqldemo.cc 1–7
proesql.h 1–7

-fl Preprocessor parameter 3–10

flag ESQL-LIB argument
defined 4–3

-flist Preprocessor parameter 3–10

FLOAT data type
Progress/SQL 2–16

Full large clients
defined 1–14
functional layers 1–15

Function names
ESQL applications 2–9

Function returns
exceptions 4–8

Function support 1–4

G

GROUP BY clause
parameter markers 2–72

H

-h startup parameter 2–38, 3–22

HAVING clause
parameter markers 2–71

Help
xvii

HLC in schema triggers 2–15
See also Schema triggers

Host language variables 2–21
data types 2–22
declaring 2–23
dynamic ESQL 2–21
dynamic FETCH statement 2–84
dynamic OPEN statement 2–82
ESQL-LIB 2–21
EXECUTE statement 2–83
indicator variables 2–25
parameter markers 2–76
rules for using 2–23

Hyphenation control
preprocessing applications 3–11
running applications 3–18

Hyphens
in ESQL 2–12

I

Identifiers in Progress/ESQL 1–4
allowing hyphens 3–11

Immediate execution
compared to prepared execution
2–53
defined 2–49
using 2–51

IN predicate
parameter markers 2–72

INDICATOR member of
SQLVAR_T 2–62

Indicator variables 2–24
declaring 2–25
exchanging null values 2–25
host language variables 2–25
specifying 2–25
truncated strings 2–24

INSERT statement
parameter markers 2–71

Installation
directory 1–7

INTEGER data type
Progress/SQL 2–15

INTO clause
FETCH statement 2–84

Invalid 4GL in schema triggers 2–14

Italic typeface
as typographical convention xiii

K

Keystrokes xiii

L

Large client executables
defined 1–5
linking 3–14
preparing to run 3–17
running 3–17
types 1–14

Large client model
configurations 3–3
CTRL-C interrupts 2–46
DataServers 3–6
defined 1–5
features 3–2
HLC in schema triggers
including in application 3–16
valid usage 2–15
process layering 3–2
small client differences 1–6, 3–6
structure 3–2
user sort routines 3–6

LENGTH member of SQLVAR_T 2–63
data type conversion B–7

Link scripts
creating 3–14

Linker response files
MS-Windows 3–15

Linking
ESQL applications 3–14

Linking ESQL/C executables 1–14, 3–14
types of executables 3–14

Lock management 2–46
callback function 2–46
CTRL-C interrupts 2–46
default 2–46
sqlsetlogoption() 2–46

Locks
default behavior 2–46

Logging messages 3–18

LOGICAL data type
Progress/SQL 2–17

M

Managing transactions
ESQL 2–40

Manual
organization of xi
syntax notation xiii

Messages
displaying descriptions xvii

Migration. *See also* also V6 to V7 migration

Monospaced typeface
as typographical convention xiii

MS-Windows
compiling ESQL/C 3–13
demo applications 3–27
linker response files 3–15
linking small clients 3–14
proesql.h 3–13
starting OIDs 3–24
woidrvr.exe command 3–24

Multi-user applications 2–44

N

NAME member of SQLVAR_T 2–64

Name qualifiers 2–39

No data outcome 4–8

NO-LOCK option 2–44
SQL requests 2–45

Non-cursor specification requests
Dynamic parameter markers 2–55

Normalized function returns 4–7

Null values
exchanging
indicator variables 2–25
indicator variables 2–24
retrieving
using indicator variables 2–26
setting
using indicator variables 2–26
SQL operators 2–12

NULLABLE member of SQLVAR_T 2–64

Number of Databases (-h) startup parameter 2–38
OID startup 3–22

O

OIB. *See also* Open Interface Broker

OID Connection (-SV) startup parameter 3–21

OID. *See also* Open Interface Driver

Open Interface Architecture
defined 1–6

Open Interface Broker
defined 1–6
shutdown 3–27
starting on UNIX 3–23

Open Interface Driver
auto-starting local 3–26
connecting to small client 3–21
connection
parameter files 3–26
Connection startup parameter 3–21
defined 1–5
functional layers 1–15
shutdown when detached 3–27
specifying client session parameters 3–22
starting manually on UNIX 3–24
starting on MS-Windows 3–24
UNIX 3–21

OPEN statement
Dynamic 2–81
cursor specifications 2–68
non-cursor specification requests 2–55
using with parameter markers 2–75
dynamic
cursor specifications 2–54
prepared execution 2–53

Operators
SQL 2–10

Options 2–2

ORDER BY clause
parameter markers 2–72

Organizing OID and database connections 3–26

P

-p Preprocessor parameter 3–12

Parameter File (-pf) startup parameter
OID and database connections 3–26

Parameter markers. *See also* Dynamic parameter markers

pcursnam ESQL-LIB argument
defined 4–3

-pf startup parameter
OID and database connectsion 3–26

-pp startup parameter 3–24

prcfile ESQL-LIB argument
defined 4–3

PRECISION member of SQLVAR_T 2–64
data type conversion B–6

Predicates
parameter markers 2–70

PREPARE statement 2–78
cursor specifications 2–54, 2–66
non-cursor requests 2–55
prepared execution 2–53

Prepared execution 2–49
compared to immediate execution 2–53
cursor specification 2–54
data exchange
using host language variables 2–53
using SQLDAs 2–53
elements 2–53
non-cursor specification requests 2–55
parameter markers 2–55
two forms 2–54
using 2–52

Prepared execution statements 2–78

Preprocessing ESQL source files 1–11

Preprocessing Progress/ESQL 3–7

Preprocessor. *See also* SQL Preprocessor

Prerequisites
using ESQL/C 1–2

Privileges
checking run-time 2–41
field 2–42
table 2–42
target for return value 2–42
user list 2–42

PROBUILD utility
creating link scripts 3–16
ESQL/C products 3–14
SQL Preprocessor 3–8

proesql.h file 1–7
using function prototypes 3–13

Progress functions
supported in ESQL 1–4

Progress lock phrases 2–44
SQL requests 2–45

-progress Preprocessor parameter 3–12

Progress/ESQL
basic elements 2–9
compared to ESQL-LIB 2–2
defined 1–1
direct use
conditions requiring 2–1

EXEC SQL statement prefixes
2–9

Progress/SQL 1–3

prooibrk command 3–23

prooidrv command 3–24

PROPATH (-pp) startup parameter
3–24

proshut command
shutting down an OIB 3–27

Q

-Q startup parameter 3–19

-Q2 startup parameter 3–19

-qf Preprocessor parameter 3–11

-qflagger Preprocessor parameter
3–11

R

R-code
including in PROPATH 3–17
portability 3–9
preprocessor generation 3–9

REAL data type
Progress/SQL 2–17

Record locks 2–44
SQL requests 2–45

Reference entries
ESQL-LIB 4–9

Request handles
retrieving for cursor requests
2–67

Request strings
defined 2–47
PREPARE statement 2–79

Requests
SQL 1–5

Reserved words
ESQL 2–9

Retrieving null values
indicator variables 2–26

ROLLBACK WORK statement
2–41

Running ESQL/C executables 3–17
small clients 3–20

Running small clients 3–20

Run-time large clients
defined 1–14
functional layers 1–15
preprocessor 3–9

Run-time managers
defined 1–16

S

SCALE member of SQLVAR_T
2–63
data type conversion B–6

Schema triggers
accessing 2–13
compiling 3–17
including in PROPATH 3–17
invalid 4GL keywords 2–14
large clients and HLC
including in application 3–16
valid usage 2–15

SELECT statements
parameter markers 2–72

Sessions for ESQL 2–2

Setting null values
using indicator variables 2–26

SHARE-LOCK option 2–44
SQL requests 2–45

Shutting down a detached OID 3–27

Shutting down an OIB 3–27

Shutting down ESQL/C executables
  small clients 3–27

Singleton SELECT
  Dynamic ESQL 2–48

Small client executables
  connecting to OID 3–21
  database connection 3–25
  defined 1–5
  linking 3–14
  linking on MS-Windows 3–14
  linking on UNIX 3–16
  preparing to run 3–17
  running 3–20
  shutting down 3–27
  startup syntax
    local 3–26
    remote 3–24

Small client model
  CTRL-C interrupts 2–46
  DataServers 3–6
  defined 1–5
  large client differences 1–6, 3–6
  local configurations 3–6
  networking configurations 3–5
  optimal response 3–5
  process layering 3–4
  structure 3–3
  user sort routines 3–6

Small client models
  networking configurations 3–4

Small client startup command
  remote 3–24

Small clients
  functional layers 1–15

SMALLINT data type
  Progress/SQL 2–16

Sort routines
  with ESQL/C 3–6

Specifying application names 2–37

SQL
  requests 1–5

SQL compared to ESQL 1–3

SQL Compiler
  defined 1–15

SQL data types 2–15

SQL descriptor areas
  ALLOCATE DESCRIPTOR
    statement 2–79
  data type conversion B–5
    input to parameter markers
      B–14
    LENGTH member of
      SQLVAR_T B–7
    output from columns B–8
    PRECISION member of
      SQLVAR_T B–6
    SCALE member of
      SQLVAR_T B–6
    TYPE member of
      SQLVAR_T B–5
  data type management 2–64
  DEALLOCATE DESCRIPTOR
    statement 2–86
  defined 2–53
  DESCRIBE statement 2–80
  described 2–56
  Dynamic CLOSE statement 2–86
  Dynamic FETCH statement 2–84
  Dynamic OPEN statement 2–81
  ESQL-LIB and 4–3
  EXECUTE statement 2–83
  functions 2–57
  memory map 2–60
  SQLDA_T structures
    described 2–61
  SQLVAR_T structures
    described 2–62
  structure 2–56

SQL functions
  parameter markers 2–71

SQL operators 2–10

Null values 2–12

SQL Preprocessor
application name 3–11
database connection 3–12
described 1–11
not precompiled 1–12
precompiled 1–13
ESQL-LIB calls 3–8
generating readable Static requests 3–10
modes of operation 3–7
multiple source files 3–10
non-precompile mode 3–9
output 1–14
precompile mode 3–8
database connection 3–8
DDL statements 3–8
error handling 3–9
r-code generation 3–9
specifying 3–12
PROBUILD 3–8
syntax and operation 3–10
V6 to V7 migration C–2

SQL request handles
defined 4–2
retrieving from Progress/ESQL 2–75
using with ESQL-LIB 2–77

SQL requests
executing Dynamic 2–48
PREPARE statement 2–79
record locks 2–45
specifying Dynamic 2–47

SQL schema
user names 2–20

SQL_STMTCLOSE value
parameter markers 2–78

SQL_STMTDROP value
parameter markers 2–78

sqlald()
described 4–10
SQL descriptor areas 2–58
using for cursor specifications 2–67
using with parameter markers 2–77

sqlalloc()
described 4–11
SQL descriptor areas 2–58
using for cursor specifications 2–68

sqlapplname()
described 4–13
using 2–37

sqlbool data type 2–22
returning CAN-DO values 2–42

sqlcdbind()
described 4–15

sqlclose() C–4

SQLCODE variable 2–27
ESQL-LIB 4–2
return values 2–28
using in an application 2–29
WHENEVER statement 2–18

sqlcommit()
described 4–17

sqlconn()
described 4–19
using 2–38

sqlcpp. *See also* SQL Preprocessor

sqlctlc()
large client interrupts 2–46

SQLDA. *See also* SQL descriptor area

SQLDA_T structures
described 2–61
See also SQL descriptor areas

sqldate data type 2–22

sqldec data type 2–22

sqldelrow()
described 4–20

sqldinit()
described 4–22
SQL descriptor areas 2–58

sqldiscon()
described 4–24
using 2–39

sqldld()
described 4–25
SQL descriptor areas 2–59
using for cursor specifications 2–69

sqldsize()
described 4–27
SQL descriptor areas 2–58

sqldynclose()
described 4–29
using with parameter markers 2–78

sqldyndesc()
described 4–32
SQL descriptor areas 2–58
using for cursor specifications 2–54, 2–68
using with parameter markers 2–77

sqldyndescinp() 2–75
described 4–34
ESQL-LIB 2–77
SQL descriptor areas 2–58

sqldynexec()
described 4–36
prepared execution 2–53
using for cursor specifications 2–54
using for non-cursor specification requests 2–55
using with parameter markers 2–77

sqldynftch()
described 4–38
using for cursor specifications 2–54
using with parameter markers 2–77

sqldyngetn()
described 4–40

sqldynncols()
described 4–42
SQL descriptor areas 2–58
using for cursor specifications 2–67
using with parameter markers 2–76

sqldynnparms() 2–75
described 4–44
ESQL-LIB 2–77
SQL descriptor areas 2–58

sqldynopen() C–4

sqldynposfetch()
described 4–46

sqldynprep()
described 4–49
prepared execution 2–53
using for cursor specifications 2–54
using for non-cursor requests 2–55
using with parameter markers 2–76

sqldynrq()
described 4–52
using 2–52

sqldynsetn()
described 4–54
using for cursor specifications 2–54
using with parameter markers 2–77

sqlfetch()
described 4–56

sqlfmt[] array
  defined 4–3

sqlfree()
  described 4–58
  SQL descriptor areas 2–59
  using for cursor specifications 2–69

sqlgetmsg()
  described 4–60

sqlgetrqhdl()
  described 4–62
  parameter markers 2–74, 2–75
  using for cursor specifications 2–67

sqllogin()
  application names 2–37
  described 4–64
  specifying parameters 2–35
    techniques 2–35
  using 2–34

sqllogout()
  described 4–67
  using 2–37

SQLN member of SQLDA_T 2–61

sqlnprepd() C–4

sqlprepd() C–4

sqlrequest()
  described 4–68

sqlrollback()
  described 4–71

SQLRQHDL data type 4–2

sqlrtnmem() C–4

sqlsetlogoption()
  described 4–72
  using 2–46

SQLSTATE variable 2–29
  ESQL-LIB 4–2
  return codes 2–30
  using in an application 2–33

sqltxt[] array
  defined 4–3

sqlupdrow()
  described 4–78

sqlv[] array
  defined 4–3

SQLVAR[1] member of SQLDA_T 2–61

SQLVAR_T structures
  described 2–62
  See also SQL descriptor areas

Starting an ESQL session 2–34

Starting ESQL/C executables
  large clients 3–17
  small clients
    basic steps 3–20
    local 3–26
    remote 3–24

Starting small clients
  basic steps 3–20
  local 3–26
  remote 3–24

Startup parameters
  common ESQL/C 3–18
  OID connection list 3–21
  SQL Preprocessor 3–10

Statement identifier, Dynamic
  associating with cursor 2–81
  defined 2–79
  open cursors 2–76
  scope 2–67, 2–79
  SQL request handles 4–2

Statements for prepared execution 2–78

Static data type conversion B–3

Static Embedded SQL 2–9

compared to Dynamic ESQL 2–2
data type conversion B–3
defined 1–2
generating readable requests 3–10
variables and constants 4–3

Static ESQL. *See also* Static Embedded SQL

Stopping ESQL sessions 2–37

Stopping small clients 3–27

-SV startup parameter 3–21

Syntax
allowing parameter markers 2–70
ESQL 2–9
forbidding parameter markers 2–72

Syntax notation xiii

T

TCP/IP
small client startup 3–24

Transaction management 2–40
DDL statements 2–41
logging out 2–40

Truncated strings
indicator variables 2–24

TYPE member of SQLVAR_T 2–63
data type conversion B–5

Typographical conventions xii

U

UNIX
compiling ESQL/C 3–12
prooibrk command 3–23
prooidrv command 3–24
starting OIBs 3–23
starting OIDs manually 3–24

UPDATE statement
parameter markers 2–72

USING clause
DESCRIBE statement 2–80
Dynamic OPEN statement 2–81
EXECUTE statement 2–82

V

V6 to V7 migration C–1
DESCRIBE statement C–2
ESQL-LIB C–3
building executable C–4
exit() C–2, C–3
function return behavior C–3
host language variables C–2
SQL Preprocessor C–2
building executable C–3
sqlcdbind() C–3
sqlclose() C–4
sqldynopen() C–4
sqlfetch() C–3
sqllogout() C–2, C–3
sqlnprepd() C–4
sqlprepd() C–4
sqlrequest() C–3
sqlrtnmem() C–4
sqlupdrow() C–3
sqlv[] references C–3

Variables
Static ESQL 4–3

W

wesqldmo.cc file
building and running 3–27

WHENEVER statement 2–18

WHERE clause
parameter markers 2–70

WIPC

OID direct startup 3–24
WITH MAX clause 2–79
woidrvr.exe command 3–24
Writing an application
ESQL/C 2–3