Power Programming with SQLWindows®

Power Programming with SQLWindows®

GUPTA™

Rajesh Lalwani

Trademarks

Quest, SQLBase, SQLGateway, SQLRouter, SQLHost and SQLTalk are registered trademarks of Gupta Corporation. SQL/API, SQLNetwork, SQLConsole, QuickObjects, Fast Facts, Gupta and the Gupta Powered logo are trademarks of Gupta Corporation. SQLWindows, is a registered trademark and TeamWindows, ReportWindows and EditWindows are trademarks exclusively used and licensed by Gupta Corporation.

IBM and IBM PC are registered trademarks of International Business Machines Corporation. AS/400, Database Manager, DB2, OS/2, Presentation Manager, and Token-Ring are trademarks of International Business Machines Corporation.

Microsoft and MS-DOS are registered trademarks of Microsoft Corporation. Windows is a trademark of Microsoft Corporation.

GIF and Graphics Interchange Format are trademarks of Compuserve, an H&R Block Company.

Informix is a registered trademark of Informix Software, Inc. INGRES is a trademark of Ingres Corporation. ORACLE is a registered trademark of Oracle Corporation. SYBASE is a registered trademark of SYBASE, Inc.

All other brand and product names are trademarks or registered trademarks of their respective owners.

Copyright

Copyright © 1994 by Gupta Corporation. All rights reserved.
Power Programming with SQLWindows, 20-2235-1004
July 1994
Author: Rajesh Lalwani

Technical Editing: Ben Steverman, Client Server Systems

Additional Copies

Additional copies of this book may be purchased directly from Prentice Hall.

Specify number of copies. Use Title Code 10861-3.

Prentice Hall
PTR Division
Box 11073
Des Moines, IA 50381-1073

To order by phone:
Call (515) 284-6751, or
FAX to: (515) 284-2607

Quantity Orders: For purchase of more than 30 copies to be mailed to a single address, please write to Prentice Hall, Corporate Sales Dept., Englewood Cliffs, NJ 07632 for information on discounts or please FAX to (201) 592-2249.

Dedication

To my family:

Parents—Lata and Bhagwandas
Brothers—Umesh and Mahesh
My wife, my friend—Sunita

Foreword

Power Programming with SQLWindows documents the SQLWindows promise: quick and powerful application development. Building on the fast start and high productivity provided by Gupta's new QuickObjects architecture, *Power Programming* is the best one-volume tutorial for extending or creating QuickObjects and building unrivaled robustness with SAL, the SQLWindows Application Language.

At Gupta, our vision for application development is a world of objects, created and assembled by teams of developers working together to build a new generation of easy, flexible, networked applications. Today, SQLWindows 5 starts you on the path to a team-based, object-oriented world with a foundation of components for the collaborative modeling, construction, assembly, management and execution of powerful client/server applications. With support for external CASE tools, a design environment for creating and assembling QuickObjects, a repository for managing libraries and supporting teams and the industry's first 4GL compiler, only SQLWindows fully equips you for rapid application development in the coming world of objects.

Power Programming with SQLWindows is the best and fastest way to tap all the power of SQLWindows 5. It condenses in one book the major programming principles embedded in the extensive documentation that accompanies the Starter, Network and Corporate Editions of SQLWindows 5. This volume also provides the detail documentation for new SQLWindows Solo, Gupta's quickest route to client/server familiarity. With SQLWindows 5 and *Power Programming,* I am confident you'll be building groundbreaking client/server applications as fast as you can conceive them.

Umang Gupta
President and CEO
Gupta Corporation

Contents

Foreword vii

Contents ix

Preface xxi

Acknowledgments xxiii

1 **Introduction** 1

About Gupta Corporation 1
SQLWindows 5 and the Gupta Client/Server Suite 1
What's in SQLWindows 5? 3
The Power to Get Client/Server Done 8

2 **Getting Started with SQLWindows** 9

Typical SQLWindows Applications 9
Tool Palette 11
Customizer 11
QuickObjects 12
Building Your First SQLWindows Application 15
About QUICKFRM.APP 15
Creating a New Application 16
Selecting Window Type and Toolbar Position 17
Adding a Database 17
Specifying a Data Source 18
SQLWindows Creates the Complete QuickForm 19
Changing Attributes of the Form Window 19
Taking a Break 19
Running an Application 20
Using QuickObjects 20
Creating a New Application 21
Placing the Data Source on the Form Window 21

Making the Data Source Invisible 23
Placing Visualizers on the Form Window 23
Defining Properties of a QuickObject 24
Placing Commanders on the Form Window 25
Running QUICK.APP 25
Writing Your Own Code 25
Application Outline 26
Outline Views 27
Application Libraries 28
Outline Options 29
Debugging an Application 31
Animation 32
SQLWindows Compiler 32
Creating an Executable (.EXE) File 32
Creating a Program Item in Program Manager 33
Distributing Your Application 34
Release 4.1 34
Release 5.0 34

3 Building a Database Application 37

About DATABASE.APP 37
Application Actions 38
Defining Variables 42
SAM_AppStartup 42
Displaying the Login Dialog Box 43
SAM_Create 44
Using Resources 45
hWndItem 45
Connecting to the Database 46
SAM_SqlError 46
SqlGetRollbackFlag 47
Displaying Error Text, Reason, and Remedy 47
Ending the Dialog Box 50
DBP_PRESERVE—Cursor Context Preservation 50
Isolation Levels 51
The Main Form Window frmMain 54
Keeping Track of Changes 54
SAM_Validate 54
SAM_Validate or SAM_AnyEdit? 55
SalPostMsg or SalSendMsg? 55

SAM_Close 59
Creating Result Set....60
Preparing (Compiling) a SQL Statement....61
Executing a SQL Statement....61
ROWID....62
Fetching First Row....62
Fetching Next Row....64
Fetching Previous Row....64
Fetching the Last Row....65
Deleting a Record....66
Deleting a Record in a Multi-user Environment....66
Inserting a New Record....69
Finding All Children....70
SalGetFirstChild....70
SalGetNextChild....70
SalClearField....70
Committing Changes....70
Inserting a New Record....72
Calculating Next Sequential Number....72
SqlImmediate....73
Updating an Existing Record....73
Updating a Record in a Multi-User Environment....74
Undoing (Discarding) Changes....75
Exiting from the Application....76
Named Transactions....76
SqlConnectTransaction....77
SqlSharedSet....77
SqlSharedAcquire....77
SqlSharedRelease....78

4 Object-Oriented Programming 79

Software 'Manufacture'....79
Class – Base Component of OOP....80
Designing Frequently Used Classes....80
Auto Entry Data Field Class—clsDfAutoEntry....80
SalGetMaxDataLength....81
Instance Variable—nMaxLength....82
SAM_AnyEdit....82
SalStrLength and SalStrGetBufferLength....82
MyValue....82

SAM_SetFocus ... 83
Listing a Data Field Class in the Tool Palette ... 83
Data Field Class of Data Type Number—clsDfNumber ... 84
Defining New Classes from Previously Defined Classes ... 84
Deriving New Classes from Multiple Base Classes ... 85
Parts and Assembly Case Study—Browse Screen ... 86
Designing Parts ... 86
Sql Handle Class—clsSqlHandle ... 86
Local Error Processing—When SqlError Statement ... 92
Sql Handle Class for SELECT statements—clsSqlHandleSelect ... 92
Form Window Class for Browse Screens—clsFrmBrowse ... 97
hWndForm ... 99
SAM_Destroy ... 99
Assembly Line ... 103
Late Binding versus Early Binding ... 105
Corporate Standards for User Interface ... 107
Easy Maintenance of Code ... 107
Hiding Implementation Details ... 108

5 MDI Windows 109

About MDI Windows ... 109
Managing Phone Numbers and Addresses ... 109
Structure of PHAD Table ... 111
Architecture of an MDI Window Application ... 112
Architecture of PHAD.APP ... 112
Application Global Declarations ... 115
External Functions ... 115
Dynamic Link Library (DLL) ... 117
When to Use DLLs ... 117
Declaring External Functions ... 118
Calling External Functions ... 119
User Constants—Defining Programmer Messages ... 119
Named Menus ... 120
Predefined Named Menus ... 121
Global Variables ... 122
Class Definitions ... 122
clsIndexPushButton ... 126
General Window Class ... 127
clsDfResetDirty and clsMlResetDirty ... 127
clsSqlHandleSelectPhAd ... 127

clsSqlHandlePhAd 128
Application Actions 128
Dialog Box to Report a SQL Error 129
Radio Buttons 132
MDI Window 132
SalSendMsgToChildren 137
pbZoomIn 137
Who has the focus? 137
Mnemonics 138
Manipulating a Child Window's Visibility State 138
SalGetWindowState 139
SalBringWindowToTop 139
ShowWindow 139
Index Push Buttons 141
PM_IndexLetter 141
SAM_Close 141
Functions 142
Accessing System Menu 142
GetSystemMenu 142
EnableMenuItem 142
Form Window 143
SAM_Destroy 144
Menu Item—Enabled When? 144

6 **Table Windows** **159**

About Table Windows 159
Types of Table Windows 159
Lines Per Row 160
Allow Row Sizing 161
Word Wrap 161
Cell Type 161
Table Window Cache 161
PHAD.APP—the Table Window 164
Defining the Table Window 164
Initialization—On SAM_Create 166
Populating the Table Window—SalTblPopulate 167
SalTblPopulate 167
Methods to Populate a Table Window 168
Table Window Flags 169
Browsing Through Rows 171

SalTblSetRow ... 171
Focus Row ... 171
Selected Row ... 171
Row Flags ... 172
Context Row ... 172
SalTblFetchRow ... 173
SAM_Click ... 174
SAM_RowHeaderClick ... 174
Row Validation—SAM_RowValidate ... 175
Searching among Table Window Rows ... 176
Binary Search ... 176
SalStrUpperX ... 176
Using Column Names as Variables – Setting Context ... 177
SalTblQueryFocus ... 178
SalTblSetContext ... 179
Maintaining Records Using a Table Window ... 179
Application Global Declarations ... 179
Table Window tblMain ... 181
How to Check If There are Modified Rows ... 184
SalTblAnyRows ... 184
Populating the Table Window ... 185
Marking a Row for Deletion ... 186
SalTblSetFlagsAnyRows ... 187
Inserting a New Row ... 187
SalTblInsertRow ... 188
SalTblSetFocusCell ... 188
Applying Changes—Updating the Database ... 189
Order of DELETE, INSERT, and UPDATE Operations ... 189
SalTblDoDeletes ... 191
SalTblDoInserts ... 191
SalTblDoUpdates ... 192
Discarding Changes ... 193
Disconnecting from the Database ... 194

7 Generating Reports 195

The Process of Generating a Report ... 195
Designing a Report Template for an Invoice ... 197
Creating a New Template ... 198
Defining Input Items ... 199
Defining a Break Group ... 199

Input Totals....201
Defining ItemNum Input Total....201
Name....201
Formula....201
Statistic....202
Restart Event....202
Pre-Process....203
Defining TotalAmount Input Total....203
Defining a Formula....204
Data Items....204
Functions....204
Operators....205
Formula Name....205
The Tool Palette....206
Selector Tool....206
Auto Selector....206
Field Tool....206
Picture Tool....207
Background Text Tool....207
Box Tool....208
Line Tool....208
Page Header....209
Break Group Header for ORDER_NUM....209
Detail Block....210
Alternate Background....210
Break Group Footer for ORDER_NUM....211
Printing or Displaying a Report....211
Displaying the Report Dialog Box....212
Populating Combo Boxes of the Report Dialog Box....213
Combo Box....214
SalListPopulate....214
SAM_Click....215
Launching the Report....217
SalReportPrint....218
SalReportView....219
Feeding Data to the Report – One Row at a Time....220
SAM_ReportStart....221
SAM_ReportFetchInit....221
SAM_ReportFinish....222
Cross Tabular Report....224

Defining Input Items 226
Defining Input Cross Tab 226
Defining Rows 226
Defining Columns 227
Defining Cell Value 228
Defining Statistics 229
Defining Summary Statistics 229
Naming the Cross Tab and Specifying the Restart Event 230
Placing the Cross Tab in the Report Template 230
Fine Tuning Tabs and Choosing Landscape Orientation 231
Feeding Data to the Report 232
Mailing Labels 232
Defining Input Items 232
Defining Number of Columns 233
Placing Input Items and Formulas in Fields 233
StrLength 234
StrIFF 234
Supressing Blank Lines 235
Specifying Minimum Height for the Detail Block 236
Printing Multiple Reports in a Batch 236
Check Box 237
Number of Copies, All or Ranges of Pages 237
SalDisableWindowAndLabel 238
SalEnableWindowAndLabel 238
Launching the Next Report 241
Handling Report Messages 242

8 Creating Your Own QuickObjects 245

Creating Your Own QuickObjects 245
What is a QuickObject? 246
Named Properties 246
Process of Creating a QuickObject 247
Defining a QuickObject Class 247
Custom Interface Applications 249
QCKPROP.APP – Custom Interface Application for cMicroHelp 250
Using the QuickObject Editor 253
Application Name 253
Dialog Name 254
Launch Dialog When Dropped? 254
Palette Bitmap 254

Customizer Text 254
Show Data Sources on Palette? 254
Using cMicroHelp QuickObject in an Application 255
Deriving New QuickObjects from Existing Ones 255
SalSendClassMessageNamed 256

9 Advanced Topics 259

Pictures and OLE (Object Linking and Embedding) 259
Graphic Images 260
File Storage 261
Picture Transparent Color 261
Picture Fit 261
Tile to Parent 262
Picture Functions 262
About OLE 262
OLE Objects 262
Linking 263
Embedding 263
Client and Server Applications 264
OLE Verbs 264
Building an OLE Application—ScrapBook 264
Structure of the SCRAPBOOK Table 265
Form Window—frmMain 266
Resetting Contents of a Picture 269
SalPicClear 270
Paste—Embedding an OLE Object 270
SalEditCanPaste 270
SalEditPaste 270
Paste Link—Pasting an OLE Link 271
SalEditCanPasteLink 271
SalEditPasteLink 271
Managing OLE Links 272
SalOLEAnyLinked 273
SalOLELinkProperties 273
Performing Object Related Actions (OLE Verbs) 274
Inserting a New OLE Object 274
SalEditCanInsertObject 274
SalEditInsertObject 274
Creating a Popup Menu at Runtime 275
SalTrackPopupMenu 275

Detecting Changes to an OLE Object 276
SalOLEAnyActive 276
Field Edit Flag 277
SalQueryEditFlag 277
SalSetEditField 278
Fetching an OLE Object from a Database 278
SalPicSetString 279
SalPicGetDescription 280
Inserting or Updating an OLE Object in a Database 281
SalPicGetString 281
Compressing a String before Storing in a Database 282
SalStrCompress 282
SalStrUncompress 282
Drag and Drop 285
Dropping Files from the File Manager 285
SalDropFilesAcceptFiles 285
SAM_DropFiles 286
SalDropFilesQueryFiles 286
Drag and Drop between Application Windows 287
Source Window Messages 288
Target Window Messages 288
Drag and Drop Related Functions 289
VBX Controls 290
Placing a VBX Control 290
VBX Interface 291
Events 291
Properties 291
VBX-Related SAL Functions 292
Team Programming 295
TeamWindows 295
Repository 297
Project 297
TeamWindows Users 298
Modules 299
Module Types 299
Defining New Module Types 300
Module Storage Methods 301
Checking Out and Checking In a Module 301
Extracting a Module 302
Module Relationships 302

TeamWindows Development Levels 302
Templates 303
Impact Analysis 304

Glossary 305

Index 319

Preface

SQLWindows, first introduced in 1988, is the most widely installed client/server application development system for the Microsoft Windows environment. With the introduction of SQLWindows 5 and QuickObjects, it is clear that SQLWindows has a promising future in the years to come.

SQLWindows is an easy-to-use, yet powerful, 4GL tool to write client/server applications. On one hand, it has features such as QuickObjects that make it extremely easy to write a fully functional client/server application without writing a single line of code or with minimal programming. On the other hand, SQLWindows Application Language, SAL, is like languages such as C or C++ where the complexity of the applications developed is only limited by the creativity of the programmer.

SQLWindows provides a very rich environment for application development. It provides many ways to build applications and solve programming problems. The purpose of this book is to provide examples of some of the better ways to build applications using SQLWindows. It is my intention to present a quick and an easy way to learn SQLWindows programming – through generous use of real-life examples. This book explains:

- How to write a complete database application without writing a single line of code – using QuickObjects.
- How to build a database application from scratch – browsing records of a table, modifying or deleting an existing record, inserting a new record, and generating sequential numbers to be used as identifiers. An introduction to named transactions – new in SQLWindows 5.
- Working in a multi-user environment.
- Object-oriented programming. How to extend existing QuickObjects. How to create your own QuickObjects.
- MDI Windows, and Table Windows. I build a complete MDI application PHAD to manage phone numbers, addresses, and important dates.
- Generating reports – invoices, cross-tabular reports, and mailing labels. Printing reports in a batch.

- Advanced topics – Object Linking and Embedding (OLE), Drag and Drop, Visual Basic (VBX) Controls, and Team Programming.

I will be satisfied if reading this book makes you a more confident and efficient SQLWindows programmer. I will feel happier if it leaves you with a burning curiosity to explore the full potential of SQLWindows by looking at the extensive online help and reference manuals that come with SQLWindows.

The example applications are developed for SQLWindows 5, but most of the concepts and techniques also apply to prior versions such as SQLWindows 4.1.

It is assumed that you already know how to use Microsoft Windows. It will help if you have already seen or used some database applications, particularly ones with a graphical user interface. While not necessary, it will be easier for you to learn SQLWindows programming if you have done some event-driven programming such as for Microsoft Windows, Macintosh, or X-Windows (Motif). Finally, I assume some knowledge of relational databases, and SQL.

Throughout the book, I explain concepts by giving complete examples. Secondary concepts and new terms are explained when they are used for the first time. My goal is not only to explain the features of SQLWindows, but also to show you how they fit together. You will find generous use of listings of actual code.

While this approach works very well for a beginning programmer, it presents some challenges when later used as a reference by the same programmer or by an experienced programmer. To help solve this problem, I provide an extensive Index at the end. If you want to refer to a SQLWindows message or function, it may not be easy to find it using the table of contents in the beginning of the book. But looking at the Index will immediately point you to the appropriate page. There may be several pages listed for each entry in the Index. I have made every effort to list, in bold, the page number where you should go first.

I have organized the book and its chapters loosely around classes of applications. For example, Chapter 2 shows you how you can quickly create fully functional applications using QuickForms and QuickObjects. Before you can extend the functionality of the existing QuickObjects or create your own QuickObjects, you need to learn some fundamental concepts of SQLWindows programming such as building a database application (Chapter 3), object-oriented programming (Chapter 4), MDI windows (Chapter 5), Table Windows (Chapter 6), and

generating reports (Chapter 7). Chapter 8 comes back to QuickObjects and shows you how you can create your own QuickObjects or modify the existing ones. Finally, Chapter 9 introduces some advanced topics such as OLE, drag and drop, VBX controls, and team programming.

I have a few comments about how I wrote this book. I started writing this book with Microsoft Word for Windows 2.0. It was clear very quickly that Word 2.0 did not have the features to handle such a big project. Word 6.0 arrived in time for this book and rescued me. Screen shots were captured using DoDOT from Halcyon Software. I gave final touches to the .BMP and .GIF images using Paint Shop Pro from JASC, Inc. IconLib (Fax: 408/446-2563) provided the icons used on toolbar push buttons.

Acknowledgments

A book such as this cannot come about without help and encouragement from some very special people. I offer my sincere thanks to the following:

- Pat Pekary, Karen Gettman, Bruna Byrne, and Ann Stein from my Hewlett-Packard days for the encouragement they provided for a book on POSIX that I was planning to write for the Hewlett-Packard Press. (For the curious minds, my interests changed and I joined Gupta Corporation.)
- Brett Bartow and Paul Becker with Prentice Hall for showing confidence in me and working with me throughout the project to ensure its success.
- Gupta Corporation for providing the necessary resources to ensure the successful completion of this project in a timely manner. I would like to thank Candace Sestric, Clark Catelain, Earl Stahl, Rich Heaps, Kevin Johnson, and Umang Gupta for creating the necessary framework for this project.
- Denise Tindell at Gupta Corporation for managing the day-to-day matters of this project. She helped me schedule the project (and made sure I met the deadlines!). She removed the obstacles and provided access to the resources in different organizations at Gupta.
- Ben Steverman of Client Server Systems for reviewing the book and example applications for technical accuracy.
- Earl Stahl for reviewing the book and making sure that the book meets Gupta standards.

- Umang Gupta for sharing his thoughts with us in the Foreword.
- Phil Ressler, Vik Chaudhary, and Audrey Kalman for help with the product issues.
- K. Smokey Cormier, Denise Tindell, Susan Wilson, and Patricia Wright for working on my language and grammar.
- All the people who worked on the technical reference manuals, online help, and release notes of SQLWindows – Bruce Ring, Susan Wilson, and Patricia Wright, K. Smokey Cormier, and Denise Tindell. This book uses some material originally written by them.
- Mahesh Lalwani for pointing out the unnecessary index entries and suggesting additional ones.
- Managers in my organization – Susan Abraham, Bob Bramley, Tim Conway, Taraneh Derak, Robin Moss, and Candace Sestric for letting me work on this project.
- My colleagues—Shailesh Bhandari, Jim Cable, Mahesh Lalwani, Suman Kamal, Keith Remmes, Charity Silkebakken, Sanjay Tandan, Kuntal Thakore, and Steve Whitt. I hope they all missed me when I was away working on the book!
- Family and friends—Leela and Ramchand Lalwani, Kaushalya and Lalchand Lalwani, Bhagwanti and Amar Lilani, Lakshmi and Govind Sadhwani, Shelu and Ratan Bhatia, Meena and Mohan Rijhwani, Rupinder and Charanjit Singh, Sujata and Sunil Bopardikar, Girija and Sridhar Dasu, Lakshmi and Ramana Yerneni, and Jyoti and Raghu Dwarakanath; Sunita's family—Dr. N. Ram Khetpal, Neelam and Manohar Mandhani, Salochna and Shivaji Mandhan, Neetu and Rajendra Khetpal, and Beena and Deepak Kataria. They all help me keep my sanity and constantly remind me that there are pleasures other than computers that life has to offer!

Cupertino, California *Rajesh Lalwani*

July 30, 1994

1

Introduction

Gupta's SQLWindows provides both an easy, quick start and the power to finish industrial-strength applications. The product's roots and evolution, and that of Gupta Corporation itself, have paralleled—and in some cases anticipated—important market developments. This introduction offers a brief overview of Gupta Corporation, its products, and where SQLWindows fits in the client/server development picture.

About Gupta Corporation

Nearly a decade before corporations began widely adopting client/server architecture, Gupta Corporation laid the foundation for PC participation in a networked, client/server world. The company's founders understood that corporations would require a new computing architecture to meet competitive demands and achieve greater organizational flexibility. This new architecture had to provide broader and quicker access to corporate information, an easy-to-use interface, and tools to produce powerful and flexible applications.

Through the 1980s, Gupta Corporation brought this vision to the corporate computing world. In 1986, the company introduced SQLBase, the first database server for PC networks. Two years later, long before it was clear that Microsoft Windows would become a corporate standard, Gupta introduced SQLWindows, the first Windows-based client/server development system.

SQLWindows 5 and the Gupta Client/Server Suite

Leading with SQLWindows 5, Gupta Corporation provides a family of client/server development products known as the Gupta Client/Server Suite. In addition to SQLWindows and SQLBase, these products include enterprise connectivity software that brings centralized computing resources to the desktop. Gupta's Tools Integration for the Enterprise (TIE) strategy allows the easy integration of independently developed technologies such as CASE tools and groupware into applications created with Gupta products.

The Gupta Client/Server Suite is intended for designing and deploying client/server applications that combine the robust power of SQL databases with the cost-effectiveness of LANs and the ease-of-use of graphical PCs. Developers can choose components to meet their specific needs. The Gupta Client/Server Suite is comprised of:

- *SQLWindows*—A quick and powerful object-oriented system for developing client/server applications for Microsoft Windows and other graphical user interface (GUI) platforms. The new QuickObjects and QuickForms architecture make SQLWindows easy to use. SQLWindows' sophisticated 4GL, 4GL-to-C compiler, and true object-oriented programming give programmers the power to get the most demanding client/server applications done.
- *Quest Reporter and Quest*—*Quest Reporter* is a graphical data access, query and reporting tool for end users that provides an easy, intuitive interface. *Quest* provides the same powerful data access and reporting as Quest Reporter and provides advanced users the ability to create custom forms, create and manage SQL tables, and work with database definitions.
- *SQLBase*—A fully functional, multiuser relational database server available on several operating system platforms, including DOS, Microsoft Windows, OS/2, UNIX and NetWare. *SQLTalk* is an interactive user interface for SQLBase that allows you to enter SQL commands along with SQLTalk's own commands to manage a relational database. *SQLTalk for Windows*, popularly known as *WinTalk*, is similar to SQLTalk but runs under Microsoft Windows and has a graphical user interface for certain operations. WinTalk can also be used with other non-SQLBase databases.
- *SQLConsole*—A database administration and monitoring tool for SQLBase servers.
- *SQLNetwork*—A family of products providing connectivity to IBM DB2, Oracle, Sybase and Microsoft SQL Server, Ingres, Informix, OS/2 Database Manager, IBM AS/400, Cincom Supra, HP Allbase/SQL, and many other desktop and SQL databases through ODBC.

What's in SQLWindows 5?

SQLWindows is a quick and powerful object-oriented client/server application development system for Microsoft Windows and other GUI platforms. Programmers can write GUI applications with the point-and-click simplicity of Microsoft Windows, while taking advantage of industry-standard SQL to interact with any SQL database.

SQLWindows 5, the new release of SQLWindows, proves that a powerful client/server development system can also be easy to learn. The SQLWindows product line scales from the single developer writing standalone applications for desktop deployment to teams of developers working on complex multiuser applications accessing enterprise-wide data sources. The SQLWindows product line includes:

	Corporate	Network	Starter	SQLWindows Solo
SQLWindows with QuickObjects	✓	✓	✓	✓
Quest data management tool	✓	✓	✓	✓
SQLBase development engine[1]	✓	✓	✓	✓
Free deployment of 5 MB SQLBase engine				✓
ODBC connectivity to personal databases	✓	✓	✓	✓
QuickObjects for e-mail systems	✓	✓	✓	✓

[1] SQLBase development engine in Solo Edition stores up to 5 MB.

	Corporate	Network	Starter	SQLWindows Solo
QuickObjects for Lotus Notes	✓	✓	✓	
Visual Toolchest class library	✓	✓	✓	
ODBC connectivity to SQL databases	✓	✓	✓	
SQLRouters for Oracle, SQL Server, Informix, Ingres, and AS/400—for development	✓	✓	✓	
SQLRouters for Oracle, SQL Server, Informix, Ingres, and AS/400—for deployment	✓	✓		
SQLWindows Compiler	✓			
SQLConsole application tuning facility	✓			
TeamWindows, Gupta Open Repository	✓			
Interface to CASE tools via Gupta TIE	✓			

With Gupta SQLWindows Solo, developers can build single-user applications for desktop deployment. It's a great way to get started with the new world of client/server development. When you need additional departmental and enterprise connectivity options, higher performance, and team programming support, you can move your SQLWindows Solo applications up to the Starter, Network, or Corporate editions of SQLWindows without modification.

Here are the main features of SQLWindows:

- *Ease of Use with QuickObjects—QuickObjects* are powerful, prebuilt components you can use to build applications without knowing SQL or writing a single line of code. You don't need to be an experienced client/server or object-oriented programmer to use the new QuickObject architecture. In the next chapter, you design a complete application using QuickObjects. Later, you'll learn how to modify existing QuickObjects and create your own, using SAL (SQLWindows Application Language).

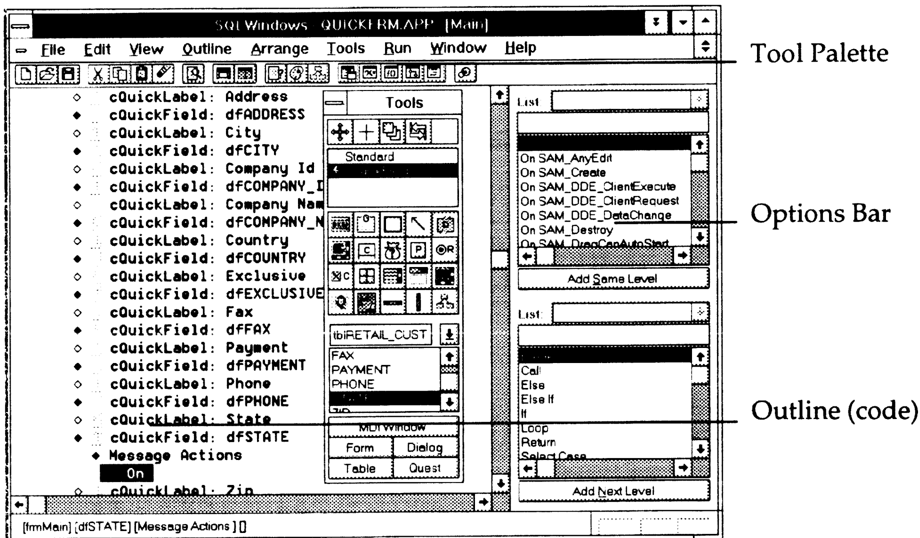

Figure 1.1 A SQLWindows session displaying outline (code), options bar, and tool palette. A QuickObject—cQuickField—is currently selected in the tool palette. This QuickObject is linked with the STATE column of the tblRETAIL_CUSTOMER table.

- *Design Window*—A visual programming environment with a rich palette of tools for designing application screens. You can simply point and click to place an object such as a push button on a form and later use drag-and-drop to move or resize it. The tool palette contains tools to create a form or MDI (Multiple Document Interface) window, QuickObjects, data fields, multi-line

text fields, push buttons, list boxes, and more. You can even place a *custom control* or a *Visual Basic (VBX) Control* using this tool palette.

The Design Window also provides a convenient *Customizer* for changing the attributes of an object. For example, you can change a data field from right-justified to left-justified or specify the name of a .BMP file to place a picture on a push button or a picture.

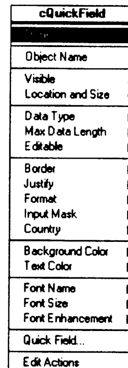

Figure 1.2 Customizer.

- *Application Outline*—A readable, collapsible outline that provides a complete overall view of the application. When you add or modify application objects in the Design Window, the application outline is automatically updated. It lets you copy, cut, and paste pieces of code as well as visual objects such forms or dialog boxes.
- *SQLWindows Application Language (SAL)*—While QuickObjects let you build a complete application without writing a single line of code, complex client/server applications usually demand that you write code. SAL is a full-featured fourth-generation language (4GL). It is object-oriented, yet doesn't require you to use object-oriented features. SAL provides many useful functions, as you will see later in the book. You can also create your own functions and, if you wish, store them in a *library*. You can even call a Microsoft Windows API function or use a function from a Dynamic Link Library (DLL) created by someone else.

- *Easy Object-Orientation*—QuickObjects give you immediate productivity while laying the groundwork for object-oriented programming (OOP) with SAL. Because QuickObjects are created on the object-oriented foundation of SQLWindows, you can use OOP techniques such as inheritance to extend the power of QuickObjects when you're ready. You can decide to use additional OOP features as you feel comfortable and as you see their value. This book discusses the OOP features of SQLWindows in Chapter 4.
- *Outline Options Bar*—This wizard-like tool is the single most useful aid for programmers writing application code. Depending on where you are in the application code, the options bar presents you with valid choices for SAL statements, available functions, variables, and constants. Point to what you want, click, and the element is automatically added to your application code. For functions, the options bar displays the data types of all the parameters to assist you. This applies not only to the standard SQLWindows functions but also to the functions that *you* have defined earlier.
- *Debugger* - The SQLWindows debugger provides multiple breakpoints, single-stepping, watch values of variables, and code animation.
- *ReportWindows*—SQLWindows' integrated, full-featured graphical report designer can produce tabular, cross-tab and control-break reports. Reports created with ReportWindows are fully compatible with reports created with Quest.
- *SQLWindows Compiler*—The SQLWindows Compiler generates C code for portions of your application when you need bare-bones performance.
- *Team Programming*—If you are developing applications in a team environment where multiple programmers work on different pieces of the code, consider using the TeamWindows feature available in the Corporate Edition of SQLWindows. TeamWindows manages the individual pieces of code, called *modules*. A module can be anything—a SQLWindows application, SQLWindows library, a .BMP file, or even a Microsoft Word for Windows document. TeamWindows provides a repository for source code and version control. The check-in/check-out facility enables team members to share and modify modules.

  TeamWindows also maintains a central data dictionary of database structural information, including table and column names and primary and

foreign-key relationships. This data can be imported from existing upper-CASE tools such as Popkin System Architect or LBMS Systems Engineer.

The Power to Get Client/Server Done

The new SQLWindows QuickObject architecture gives any developer a quick start with client/server. To extend existing QuickObjects, write your own QuickObjects, or gain more control of your business application, you need to write code. This power programming guide is your easiest way to unlock the power of SAL and discover the power to get client/server done with Gupta.

2

Getting Started with SQLWindows

The purpose of this chapter is to make you familiar with SQLWindows as quickly as possible. By the end of this chapter, you will know:

- What applications are typically developed using SQLWindows.
- Basic components of SQLWindows and how to use them. Examples of SQLWindows components are tool palette, customizer, application outline, outline options bar (or box), and full-featured debugger.
- How to write a fully functional database application using QuickObjects without writing a single line of code. Using such an application, a user can browse through records, insert new records, delete or modify existing records, and after changes have been made, either discard or apply these changes to the database.
- How to run an application.
- How to create an executable (.EXE file) from an application.
- How to distribute and deploy an executable created from an application.

Typical SQLWindows Applications

SQLWindows is an easy-to-use, yet powerful 4GL tool to write client/server applications. On one hand, it has features such as QuickObjects which make it extremely easy to write an application without writing a single line of code or with minimal programming. On the other hand, SQLWindows is like languages such as C or C++ where the complexity of the applications developed is only limited by the creativity of the programmer.

For most situations, SQLWindows provides a rich set of functions, and messages but if there is a need, a programmer can use functions from a DLL (Dynamic Link Library) including those provided by Microsoft Windows, process

Microsoft Windows Messages such as WM_CHAR, and use custom controls and Visual Basic (VBX) controls in SQLWindows applications.

SQLWindows is most often used to write GUI (Graphical User Interface) applications in a client/server environment. A typical application accesses data on a database server that supports SQL. However, SQLWindows can be used to write applications for various mail systems such as Microsoft Mail, and Lotus cc:Mail, and unstructured databases such Lotus Notes.

SQLWindows is used to write decision support applications as well as mission-critical applications. Some examples of business applications developed using SQLWindows are:

- Order entry system. A salesperson can use this system to take an order from a customer and generate an invoice.
- Patient treatment tracking system. A physician can enter the diagnosis and prescribe the treatment. The pharmacist can later use the same system to fill the prescription.
- Accounting modules. For example, accounts receivable, accounts payable, general ledger, etc.
- Payroll and human resource management system. The system may even contain pictures of employees.
- System to track sales persons and sales activities.

SQLWindows is also used in some universities to teach GUI and object-oriented programming.

Most applications are deployed with several client machines running the SQLWindows application and connected to a database server. The database server can be any database supported by SQLWindows connectivity.

As the number of people using the system grows, the server can be upgraded either by using a more powerful CPU, higher performance platform (SQLBase NLM instead of SQLBase for DOS), or a different database (DB2 running on an IBM mainframe computer instead of a SQLBase NLM running on an Intel Pentium based PC). All this can be done with little or no change to the application running on the client machines. This is a big advantage of client/server architecture.

Sometimes the entire development is done using a local, single-user edition of SQLBase server for Windows. A departmental or enterprise-wide server is used when the application is deployed.

Tool Palette

Figure 2.1 shows the SQLWindows tool palette. The tool palette is a moveable window that contains a drawing tool for each object you can create at designtime. Display the tool palette by choosing View, Tool Palette... from the

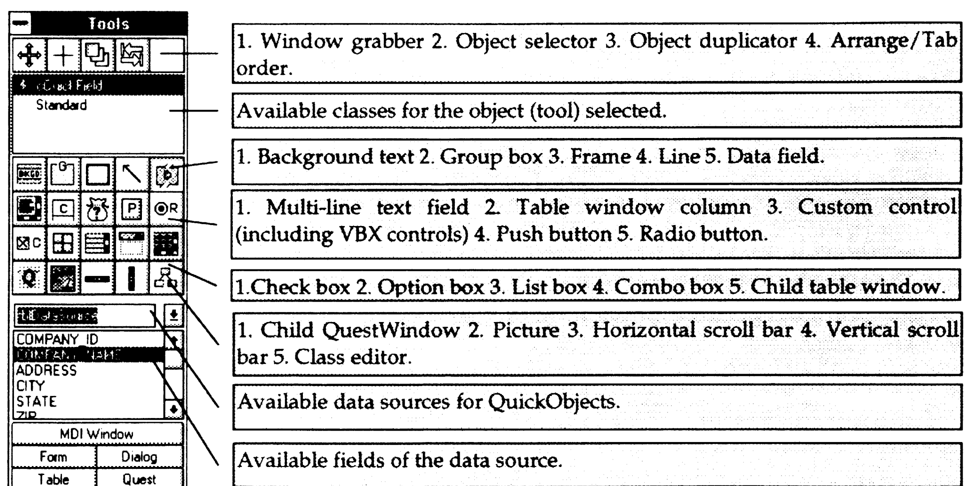

Figure 2.1 The tool palette – in wide mode.

menu or by pressing F4.

You can select a narrow or wide palette from the system menu of the tool palette.

Customizer

Each top-level and child object has attributes that define how it appears. Attributes can include the color, font, and location of the object. Object attributes can be defined via an object's customizer.

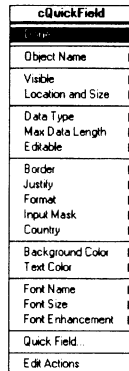

Figure 2.2 Customizer for a cQuickField data field.

There are two ways you can bring up the customizer for an object:

- Double-click or click the right mouse button on the object in the design window.
- Double-click or click the right mouse button on the box after the diamond in the outline (code) as shown in Figure 2.3. I explain application outline later in this chapter.

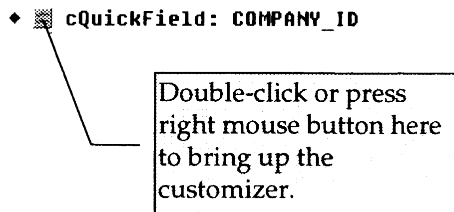

Figure 2.3 Bringing up the customizer from the outline.

QuickObjects

Before you begin writing your first SQLWindows application using QuickObjects, let me give you a brief overview of QuickObjects.

QuickObjects are a group of pre-defined objects. These objects let you develop applications in a quick and easy manner by providing a family of smart (data-aware) visual and non-visual objects.

There are three categories of QuickObjects:

- **Data Sources**

  Data source QuickObjects provide a connection to data from sources such as SQL data, email data, or Lotus Notes data. In most cases, the data source QuickObjects are made invisible after defining them. They perform the actions of accessing and updating the data behind the scene.

- **Visualizers**

  Visualizers are used to display data in objects such as radio buttons or list boxes. The type of visualizer to use depends on the semantics of the data and the user interface desired. For example, if you want to place on your form a method of payment, and there are three options—cash on delivery, credit card, and purchase order, it makes sense to use a group of three radio buttons. On the other hand, for assigning a trainer to a person in a spa, using a list box is more appropriate.

- **Commanders**

  Commanders are used to manipulate data such as viewing a previous record, retrieving data, inserting a new record, deleting an existing record, applying changes, or discarding changes. Commanders direct the data source to perform these actions. Later, you see that action displayed in the visualizers.

These three objects are typically added to a form window. Figure 2.4 shows the relationship between these categories using a SQL database data source.

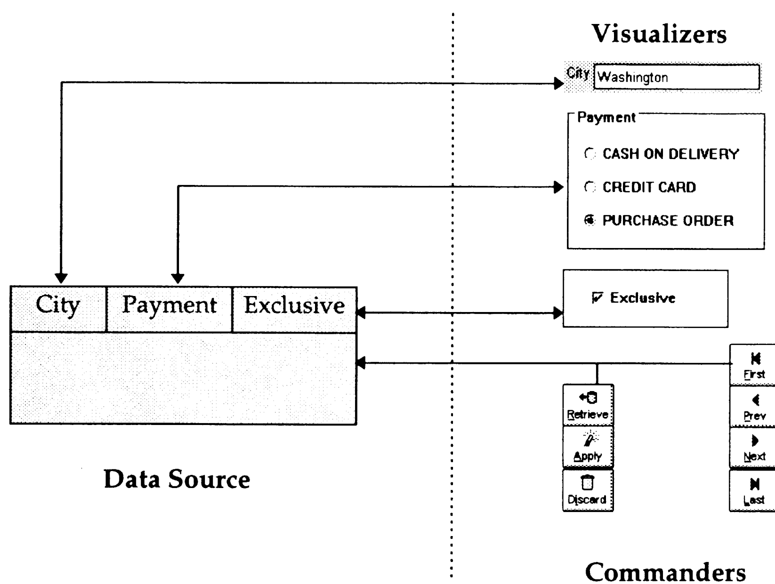

Figure 2.4 Relationship between data source, visualizers, and commanders.

Building Your First SQLWindows Application

In this section you develop and run your first SQLWindows application—QUICKFRM.APP using QuickForms and QuickObjects. I show you how to run this application. You will also create the same application by using QuickObjects directly and improve the user interface by using radio buttons and a check box instead of data fields.

Later, I explain how to generate an executable (.EXE) file for an application, and distribute, and deploy the executable version. Once you complete the process, you become familiar with the mechanical steps. In later chapters, you can focus on what goes in an application.

About QUICKFRM.APP

Figure 2.5 shows the main form window of the application. Using this application, a user can maintain records of the RETAIL_CUSTOMER table of the GUPTA database that comes with SQLWindows. The user can go to the first, previous, next, and last record, insert a new record, delete or modify an existing record. Once all the changes have been made, the user can either apply the changes to the database or discard them.

The RETAIL_CUSTOMER table contains the following columns:

```
COMPANY_ID (INTEGER, Data Required, Must be Unique)
COMPANY_NAME (VARCHAR 30)
PHONE (CHAR 15)
FAX (CHAR 15)
ADDRESS (VARCHAR 30)
CITY (VARCHAR 30)
STATE (VARCHAR 2)
ZIP (VARCHAR 10)
COUNTRY (CHAR 15)
PAYMENT (CHAR 20)
EXCLUSIVE (CHAR 3).
```

Figure 2.5 The main form window of QUICKFRM.APP.

Creating a New Application

When you start SQLWindows, it displays a dialog box as shown in Figure 2.6. If you are already in SQLWindows, choose File, New from the menu.

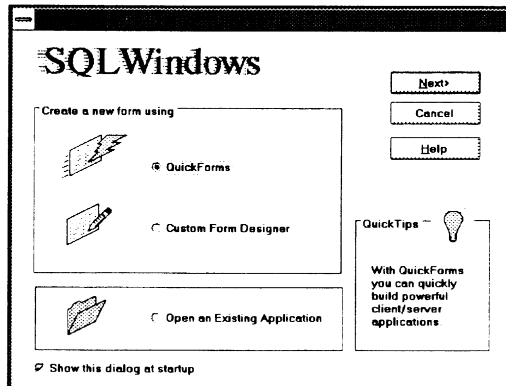

Figure 2.6 Dialog box shown when SQLWindows starts up.

At this point you can create a new application or open an existing application. For a new application, SQLWindows creates the application based on the

application specified in the preferences. Normally, it refers to the NEWAPP.APP file in the directory where you have installed SQLWindows. You can change it by choosing File, Preferences, General... from the menu.

If you choose QuickForms, SQLWindows guides you through steps to create an application using QuickObjects. QuickForms is the default, so press the Next push button to go to the next step.

Selecting Window Type and Toolbar Position

Figure 2.7 shows the dialog box displayed by the next step. You can choose to create an MDI (Multiple Document Interface) window or a form window. Choose Form Window. Using this dialog box, you can specify the position of the toolbar. The default is Top. As you can see in Figure 2.5, the form window of QUICKFRM.APP has a toolbar on the left, so choose Left and press the Next push button to go to the next step.

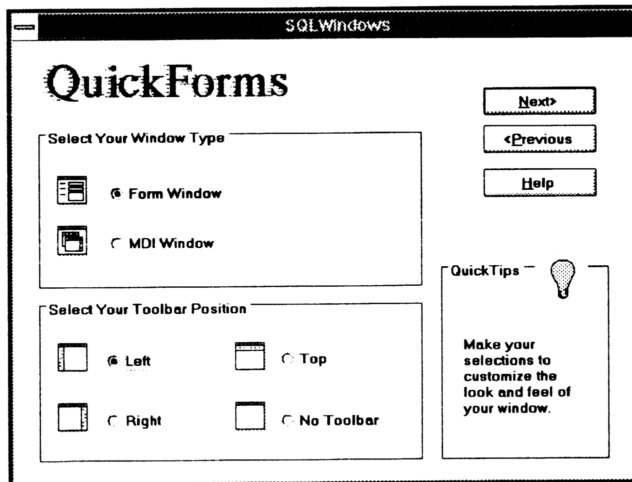

Figure 2.7 Selecting window type and toolbar position.

Adding a Database

SQLWindows asks you to specify the database name, user name, and the password that you want to use. See Figure 2.8. Specify GUPTA, SYSADM, and

SYSADM. SYSADM is the user name for the system administrator for a SQLBase database. The default password is SYSADM unless you have changed it. In this application, you only work with the GUPTA database, but if you want to work with more than one database, you can press the Database... push button to add more databases to your application.

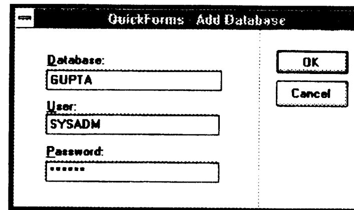

Figure 2.8 Adding a database.

Specifying a Data Source

In this step, you specify the data source. Scroll down the list on the left until you see the table RETAIL_CUSTOMER. Select RETAIL_CUSTOMER by clicking on it. Click on it again and while the left mouse button is pressed, drag it to the empty space to the right of the list box. SQLWindows shows you all the columns of the RETAIL_CUSTOMER table as shown in Figure 2.9. You can exclude a column of the table from your form by clicking on it. Since you want to include all the columns of the table, simply press the Next push button to go to the next step.

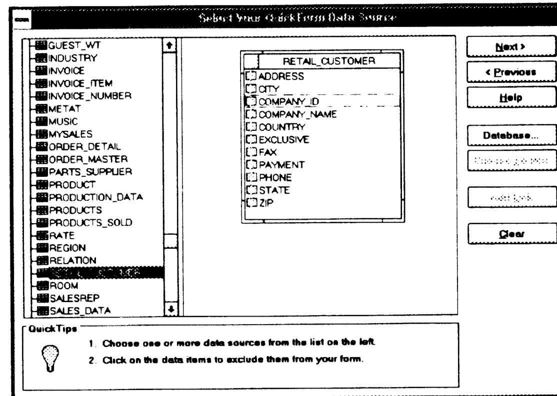

Figure 2.9 Specifying the table RETAIL_CUSTOMER as a data source.

SQLWindows Creates the Complete QuickForm

When you press the Next push button, SQLWindows creates a form window and places data fields (visualizers) for all columns of the table, along with their labels. SQLWindows also places push buttons (commanders) for browsing, inserting, deleting, etc. in the toolbar of the form window. At this point, the form window should look very similar to the one shown in Figure 2.5.

Changing Attributes of the Form Window

SQLWindows leaves you with a form window called frm1. It has a default title—QuickForm. To change the attributes, click the right mouse button anywhere on the form window to display the customizer for the form window. Do not click on the toolbar, or any child objects on the form. Choose Object Name and specify frmMain. Choose Object Title and specify Maintenance of GUPTA:RETAIL_CUSTOMER Table. When you are done using the customizer, choose Done. If you wish, you can resize the form window by grabbing the borders of the form window and dragging them with the left mouse button held down.

Taking a Break

If desired, you can take a break at this point. Choose Close from the system menu of the *form window* and choose File, Save from the *SQLWindows* menu. Exit

SQLWindows by choosing File, Exit from the SQLWindows menu. When you come back, start SQLWindows and open an existing application. If you cannot see the form window, display the form window frmMain by choosing View, Show Window... from the SQLWindows menu.

Running an Application

When building an application using SQLWindows, you are in *design mode.* There are two ways in which you can run your application. One possibility is to create an executable (.EXE) and run it from outside SQLWindows. Use this option when the application is complete and is to be distributed and deployed on end users' machines. However, when you are still in the process of developing and debugging the application, it is a time consuming process to create an executable or a runtime file, leave SQLWindows, and run it from outside SQLWindows.

The second way to run your application is from within SQLWindows itself by entering the *user mode.* You can do this by choosing Run, User Mode from the SQLWindows menu.

If there are any changes to your application since you last saved it, SQLWindows asks you if you want to save the application. I recommend saving the file every time before running the application. After saving the file, SQLWindows compiles the application and runs the application in the user mode. To enter the design mode again, you can either choose File, Exit from the menu of the application you developed, or choose Run, User Mode again from the SQLWindows menu.

Using QuickObjects

In this section I show you how to create the same application using QuickObjects. This way you can see the basic components of the QUICKFRM.APP application that you created using QuickForms. In addition, for the PAYMENT and EXCLUSIVE fields of the RETAIL_CUSTOMER table, this application (QUICK.APP) uses a group of radio buttons and a check box respectively, to improve the user interface. See Figure 2.10.

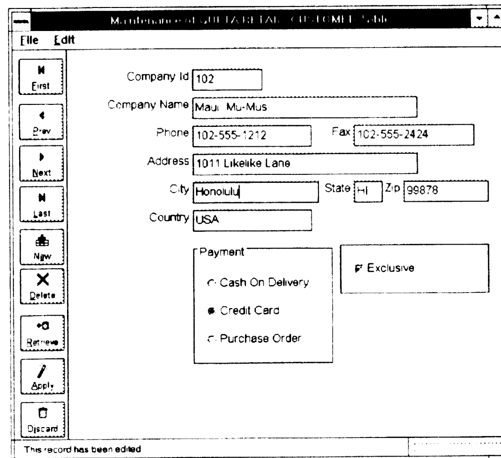

Figure 2.10 The main form window of QUICK.APP.

Creating a New Application

When you start SQLWindows, it displays a dialog box as shown in Figure 2.6. If you are already in SQLWindows, choose File, New from the menu.

Select Custom Form Designer and press the Next push button to go to the next step. This will create a new form window with no child objects on it. If you do not see the tool palette, press F4 to view the tool palette.

Placing the Data Source on the Form Window

This application uses just one data source which contains all records of the RETAIL_CUSTOMER table. To place this data source on the form window, choose the table window tool from the tool palette. When you press the table window button in the tool palette, the tool palette displays the classes available for this object in the list box towards the top of the tool palette. See Figure 2.11. Choose cQuickTable by clicking on it.

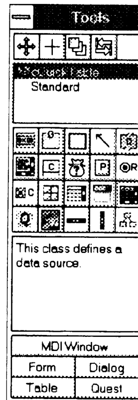

Figure 2.11 Selecting table window and cQuickTable from the tool palette.

Notice how the shape of the cursor changes to reflect the tool that you pick from the tool palette. You can place an instance of the selected object by clicking anywhere on the form window. This brings up the dialog box shown in Figure 2.12. Select the RETAIL_CUSTOMER table from the Database/Tables list box and move all the columns from the Columns list box to the list box on the right by pressing the push button with two right arrows. Press the OK push button to close the dialog box.

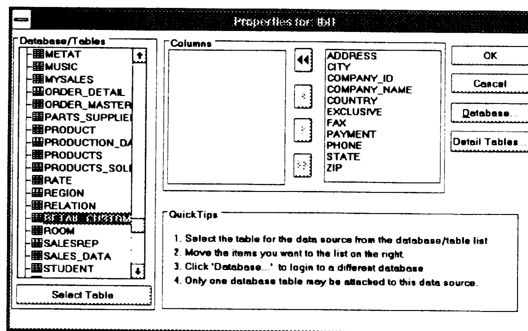

Figure 2.12 Choosing the RETAIL_CUSTOMER table and all its columns.

Making the Data Source Invisible

In most cases, you will want to hide the data source so that the data source accesses and manipulates the data behind the scene. Later, you define visualizers to view the data, and commanders to send instructions to the data source to perform desired operations.

You can hide the data source (table window) using the customizer. Open the customizer by clicking the right mouse button on the data source. Change Visible to No. Press Done to exit the customizer.

Placing Visualizers on the Form Window

In this step, you will place visualizers on the form window – one for each of the columns of the RETAIL_CUSTOMER table. See Figure 2.10 for the layout of the form window.

From the tool palette, choose the data field tool and select cQuickField from the list of available classes as shown in Figure 2.13. You will see tblRETAIL_CUSTOMER towards the bottom of the tool palette and a list of columns of the data source appears just below it. Choose COMPANY_ID because this is the first field that appears on the form window.

Place an instance of this object on the form window by clicking on the form window. Notice how SQLWindows chooses a proper size for the data field and even places a label Company Id before the data field. Repeat the same process for COMPANY_NAME, PHONE, FAX, ADDRESS, CITY, STATE, ZIP, and COUNTRY.

For PAYMENT, present the data as a group of radio buttons because there are only three choices – cash on delivery, credit card, and purchase order. It is much easier for a user to click on one of the three choices instead of typing in an entire word. Using data fields can also lead to spelling mistakes by the users.

To place a group of radio buttons for PAYMENT, choose the radio button tool, select cQuickRadioGroup, and PAYMENT from the tool palette. Clicking the left mouse button on the form window places a group of three radio buttons on the form window.

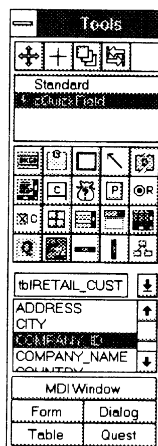

Figure 2.13 Selecting a QuickObject for the COMPANY_ID from the tool palette.

The values for the EXCLUSIVE column of the RETAIL_CUSTOMER table can only be YES or NO. Therefore, it is natural to use a check box visualizer for this column. Choose the check box tool, cQuickCheckBox, and EXCLUSIVE from the tool palette. Place it on the form window.

Defining Properties of a QuickObject

You must specify when the check box for EXCLUSIVE should be checked and when it should not. To specify this, select the check box and bring up the customizer by pressing the right mouse button. Choose Quick Check Box... to define the properties of the check box. Choose Yes and No for the Checked value and Unchecked value respectively as shown in Figure 2.14.

Finally, place a frame around the check box by choosing the frame tool from the tool palette, and placing it on the form window. To change the background color of the frame, bring up the customizer and specify Gray as the Background Color.

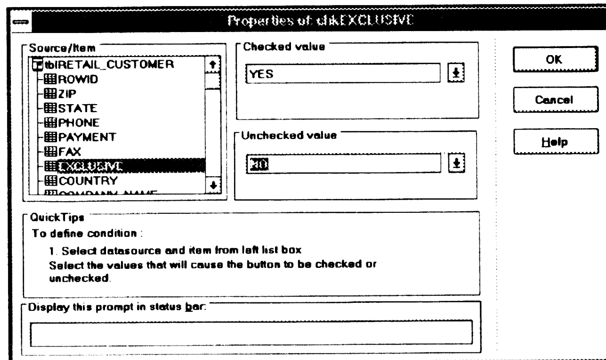

Figure 2.14 Defining properties of the check box.

Placing Commanders on the Form Window

In this step, you place commanders (push buttons) for various operations to be performed on the data source. Choose the push button tool and cQuickCommander from the tool palette. The list towards the bottom of the tool palette shows a list of all the commanders available. The names are very descriptive of what operations they perform.

You can select them one by one and place them in the toolbar of the form window. Alternatively, you can select the first commander in the list (Apply), scroll to the bottom of the list, press the Shift key, and while holding down the Shift key, select the last entry of the list (Retrieve). This selects all entries in the list box. Now, when you click the left mouse button on the toolbar of the form window, the commanders for each of the entries are placed one below the other. If you wish, move them around in the order shown in Figure 2.10.

Running QUICK.APP

You can run QUICK.APP by choosing Run, User Mode from the SQLWindows menu as discussed earlier.

Writing Your Own Code

You just developed two applications, QUICKFRM.APP and QUICK.APP, without writing a single line of code. You can develop fully functional database

applications using QuickForms and QuickObjects. As you saw, QuickObjects are smart objects – they know how to access and manipulate the data source. As you will see in Chapter 8, they are SQLWindows *classes*. You can create your own QuickObjects or even extend the existing ones. To modify existing QuickObjects, create your own QuickObjects, or to develop new applications without using QuickObjects, you need to write SQLWindows Application Language (SAL) code.

Application Outline

When you develop an application using, for example, C, you write lines of code using an editor such as emacs, vi, notepad, or Microsoft Word. For writing an application, SQLWindows provides its own application outline editor.

The application outline editor differs from a normal editor or a word processor in that it keeps the code properly structured. For example, statements to be executed after a successful If statement are at a lower level and indented towards the right side.

A useful feature of the outline editor is that you may not opt to look at levels below the level of a certain statement. See Figure 2.15. There are statements below the If statement but they are not visible at the moment. A filled diamond before the If statement indicates that there are more statements below this level. You can look at those statements by double-clicking on the diamond.

You can create a new line by pressing Insert key. You can either type the statements yourself or choose them from the outline options bar. Pressing Control-Enter continues the same statement on another line. Pressing Enter ends this statement and creates another new line. When you are done, you can press Esc to stop. You can use Alt and the arrow keys to move the highlighted statement(s) in the outline.

A filled diamond indicates that there are statements below this level.

! indicates a comment.

An empty diamond indicates that there are no statements below this level.

Status bar indicates the location of the cursor in the outline.

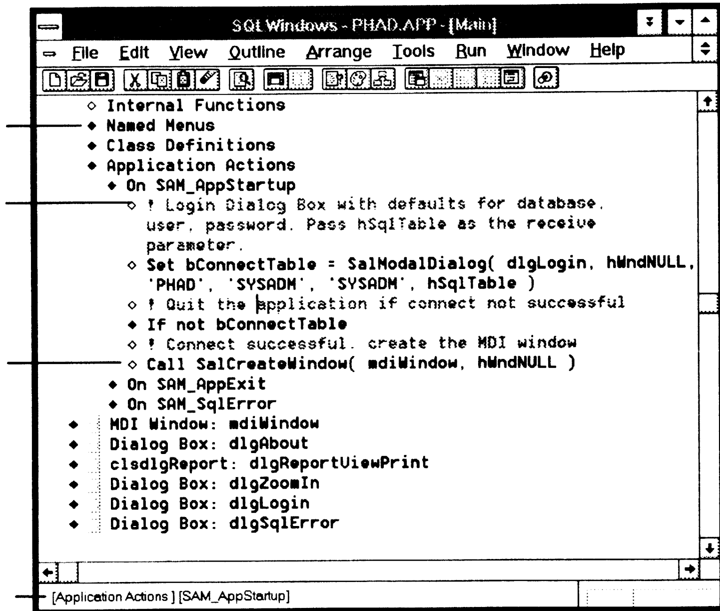

Figure 2.15 SQLWindows application outline.

Outline Views

Although there is only one outline for an entire application, you can create new views (windows) to edit specific portions. For example, you can create a new window to edit only the login dialog box dlgLogin of Figure 2.15. You can create new views by choosing Window, New Outline View... from the SQLWindows menu.

Application Libraries

SQLWindows applications are normally kept in an .APP file. Such an .APP file can include code from other files, called application libraries. Application libraries are normally kept in .APL files. There are two main reasons why one uses application libraries:

- Keep frequently used functions, dialog boxes, etc. in an .APL file so that different applications (.APP files) can include them.
- Break up a large application into several application libraries so that various programmers can work on different .APL files at the same time.

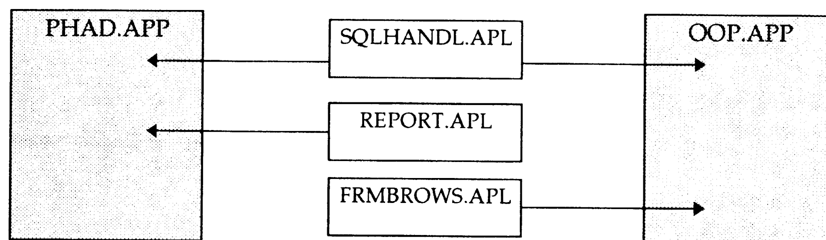

Figure 2.16 Applications (.APP files) can include application libraries (.APL files).

You create an .APL file the same way you create an .APP file. When you save it, simply choose the appropriate file type from the List Files of Type combo box and specify an .APL extension for the file name.

You can include an .APL file, for example, SQLHANDL.APL in an application by inserting a File Include: SQLHANDL.APL statement under the Libraries section. You can either specify the full pathname here or choose File, Preferences, General... from the SQLWindows menu and specify File Path(s).

You can edit an application library by opening that file in SQLWindows. Also, while you are editing an application, you can edit an included application library by bringing the cursor to a statement from that library and choosing File, Libraries, Edit Item from the menu or by pressing F5.

Outline Options

Figure 2.17 displays the outline options bar provided by SQLWindows outline editor to assist you in writing an application. To bring up the outline options bar, choose View, Outline Options... from the menu or press F2. You can also display outline options in a dialog box by choosing File, Preferences, Outline Options... from the SQLWindows menu.

The outline options bar lists items that you can add to the outline. Use the outline options bar to:

- Define menus.
- Define functions, constants, and variables.
- Code SAL statements.
- Set parameters for functions that you call.

The outline options bar is context-sensitive, as you move through the outline, the contents change to reflect the items that you can add.

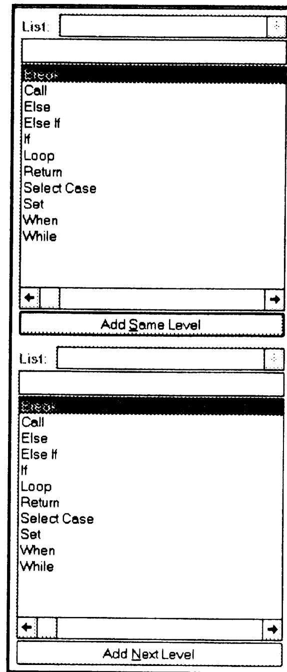

Figure 2.17 Outline options bar.

The outline options bar displays the Add Same Level and Add Next Level push buttons:

- Click Add Same Level to add the highlighted item to the application at the same level as the selected item in the outline.
- Click Add Next Level to add the highlighted item at the next level under the selected item in the outline.

Type the first few characters of an item in the data field at the top of the list boxes to scrolls to the item.

Use the combo boxes at the top to set what the Outline Options bar lists when you edit statements in the outline:

- For functions, you can list system functions, user functions, or both.

- For parameters in functions, you can list variables, system variables, constants, window names, resources, function parameters and window parameters (when positioned in a function definition or a window definition), or base classes.

Debugging an Application

SQLWindows provides a full-featured debugger that you can use to debug your code.

To set a break point on any statement, simply bring the cursor to the statement and choose Run, Break, Set from the SQLWindows menu. When you run the application by entering the user mode, execution is suspended *before* this statement, thus giving you an opportunity to examine the value of expressions and variables, and track messages.

When SQLWindows suspends the execution of the application, it displays the debugger dialog box as shown in Figure 2.18. At this point, you can evaluate an expression, continue the execution of the application, execute the current statement (step), step over the current statement, halt the application, or close the debug dialog box.

If you do not want to evaluate an expression each time the execution is halted, and want to watch a set of variables, simply press the (Watch) Variables push button and specify the variables you want to watch. You can also watch messages and the stack.

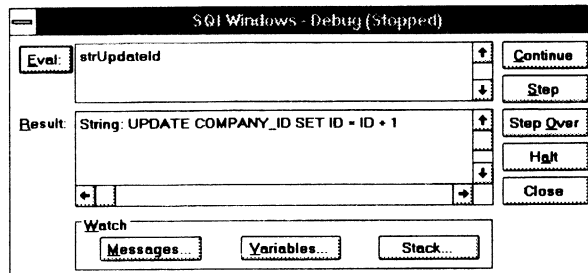

Figure 2.18 Debug dialog box displayed when the execution is suspended.

Animation

Sometimes during debugging an application, it helps to watch the statements as they are executed at runtime. This allows you to know the paths taken by the application.

Choose Run, Animate, or Run, Slow Animate from the SQLWindows menu before choosing Run, User Mode to run the application. SQLWindows highlights each item as it executes. Slow Animate does the same but at a slower speed than Animate. You can set the time interval between the execution of each item by choosing File, Preferences, General... and specifying the Slow Animate Interval in seconds.

SQLWindows Compiler

Beginning with Release 5.0, SQLWindows provides a compiler. The SQLWindows compiler improves the runtime performance and resource use of SQLWindows applications. The SQLWindows compiler:

- Translates internal functions, global variables, and constants in a SQLWindows application (source file) into C language statements and then compiles the statements into a DLL. Future versions of the SQLWindows compiler will not be limited to translating internal functions only.
- Creates a new SQLWindows application outline (output file – .APC file) that contains an external function definition for each function in the DLL. You can use this outline as an include library in applications that call the functions.

Creating an Executable (.EXE) File

When you have debugged and tested your application, and are ready to distribute the executable version of the application, you can create an .EXE file by choosing File, Make Executable... from the SQLWindows menu. This displays the dialog box shown in Figure 2.19.

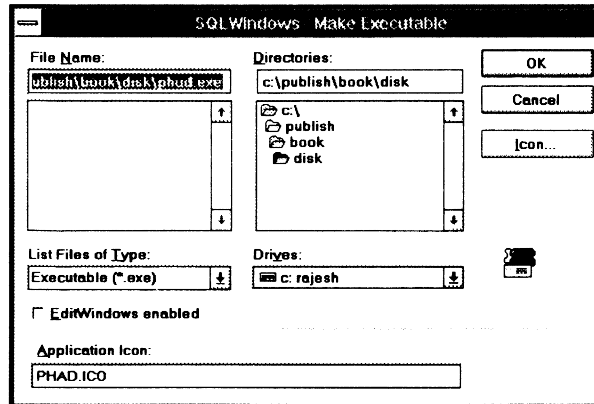

Figure 2.19 Creating an executable (.EXE) file of an application.

Use this dialog box to specify the name of the .EXE file and to specify the icon to be used with this application. (Icons are stored in .ICO files.) The companion disk includes an icon—PHAD.ICO.

Since an .EXE file does not contain the source code of your application, you can distribute your application without giving away the source code.

If you check the EditWindows enabled check box, you can later edit certain attributes of the .EXE file using EditWindows. EditWindows is mainly used to alter the .EXE file for native language support for the international market.

Creating a Program Item in Program Manager

Once you have the .EXE file for the application, you can create a program group and a program item for it so that a user can start the application by simply double-clicking on the icon.

To create a program group, in the Microsoft Windows Program Manager, choose File, New... from the menu. Choose Program Group and press OK. Enter a description for the program group in the program group properties dialog box and press OK. Choose File, New... again. This time choose Program Item and press OK. This brings up the dialog box shown in Figure 2.20.

Type in the description you want. The command line contains the name of the executable. The full path name is not necessary if the directory is in the PATH. Specify the working directory. If the directory where you have installed SQLWindows (or the necessary runtime files as discussed in the following section) is not the working directory, make sure that this directory is in the PATH. Specify the icon by pressing the Change Icon... push button. If you had specified an application icon when you made the executable, that icon appears in the list.

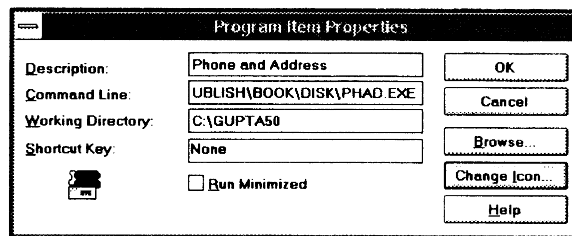

Figure 2.20 Dialog box to create a program item for an executable.

Distributing Your Application

You can distribute the .EXE file to the end users. But distributing the .EXE file alone is not sufficient as other files are needed at runtime when the end users install your application. If you are using SQLWindows 4.1 or later, all the necessary files are already placed in a separate directory called DEPLOY. If you want to verify if you have the right set of files, here is the complete list of files:

Release 4.1

```
SQLRUN41.EXE, GEE21.DLL, GRE21.DLL, GCTRL21.DLL,
GTOOLS21.DLL, RDW21.DLL, SQLNUM21.DLL, SWCSTRUC.DLL,
SQLAPIW.DLL, SWIN40.DLL, GTIOBJ21.DLL, IMAGEMAN.DLL,
IMGBMP.DIL, IMGGIF.DIL, IMGPCX.DIL, IMGTIFF.DIL,
IMGWMF.DIL, AUTOSQL.DLL, SWIN41.DLL.
```

Release 5.0

```
SQLRUN50.EXE, GEE30.DLL, GRE30.DLL, GCTRL30.DLL,
GTOOLS30.DLL, RDW30.DLL, SQLNUM30.DLL, SQLAPIW.DLL,
SWIN41.DLL, GTIOBJ30.DLL, IMAGEMAN.DLL, IMGBMP.DIL,
```

```
IMGGIF.DIL, IMGPCX.DIL, IMGTIFF.DIL, IMGWMF.DIL,
AUTOSQL.DLL, SWIN50.DLL, GCMAIL.DLL, GSW16.EXE,
GSWAG16.DLL, GSWDLL16.DLL, HPORT50L.DLL, OMS.DLL,
QCKMAIL.DLL, QCKUTIL.DLL, QGRAPH.DLL, SHRTSK30.EXE,
SQLNL.VBX, SQLSST30.DLL, SRVCAP30.DLL, VT50.DLL.
```

These runtime files do not include the SQLBase for Windows Database Engine, SQLBase Windows Client Routers, or routers for any of the other databases supported by Gupta products. These are separate products and must be purchased separately and used in accordance with the terms and conditions of the license agreement. Make sure that the SQL.INI file is installed as part of the installation of these products.

3

Building a Database Application

About DATABASE.APP

Although SQLWindows can be used for developing any graphical Microsoft Windows application, you are more likely to use SQLWindows for writing a database application—an application that accesses a database to read data, modify data, or both. As you saw in Chapter 2, you can develop fully functional database applications using QuickObjects without writing a single line of code. QuickObjects are smart objects – they know how to access and manipulate the data source. As you will see in Chapter 8, they are SQLWindows *classes*. You can create your own QuickObjects or even extend the existing ones. To modify existing (or to write your own) QuickObjects that work with database tables, you need to know how to access and manipulate database tables.

In this chapter, I will create an application, DATABASE.APP, which can be used to maintain the COMPANY table of the GUPTA database. (In SQLWindows 4.1 and earlier versions, you find this table in the SWDEMO database.) Among other things, this application shows you how to:

- Prompt the user with a dialog box to login.
- Browse through records using First, Prev, Next, and Last push buttons on a toolbar.
- Delete or update an existing record, insert a new record, and commit the transaction.
- Do delete and update operations in a multi-user environment.
- Generate sequential numbers to be used as a primary key while inserting a new record.

Figure 3.1 shows what the main form of the application looks like when finished.

In this application, I access two tables. COMPANY contains the records that this application manages. COMPANY_ID contains just one row to *remember* the next ID to use when a new record is inserted into COMPANY. It has just one column called ID, which is defined as an INTEGER. Table COMPANY has the following columns:

```
ID (INTEGER, data required)
NAME (CHAR 30)
ADDR1 (CHAR 30)
ADDR2 (CHAR 30)
CITY (CHAR 30)
STATE (CHAR 2)
ZIP (CHAR 10)
COUNTRY (CHAR 20)
PHONE (CHAR 17)
FAX (CHAR 17).
```

Figure 3.1 Main window of DATABASE.APP

Application Actions

Most applications display a login dialog box when the application starts. DATABASE.APP displays a login dialog box with default values for the database name (GUPTA), user name (SYSADM), and password (SYSADM). The user can

change any of these values and press the OK push button. If everything goes well, necessary connections are made to the database and the main form window displays. If the user name and password are not correct for the database, an error message displays. The user can type in the correct values and press OK push button again or quit the application by pressing the Cancel push button. Figure 3.2 shows the Login dialog box.

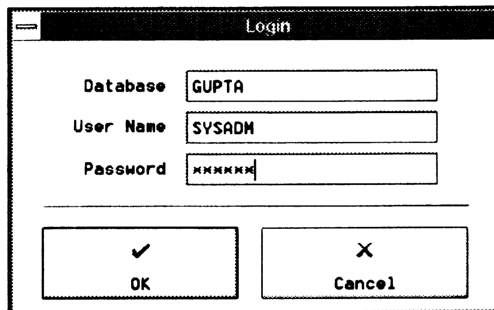

Figure 3.2 Login Dialog Box for DATABASE.APP

Let's take a look at the relevant portions of the code. Listing 3.1 shows the global variables and the Application Actions for DATABASE.APP.

```
Application Description: DATABASE.APP
  Chapter 3
  Building a Database Application
  Power Programming with SQLWindows
  by Rajesh Lalwani.
  Copyright (c) 1994 by Gupta Corporation.
  All rights reserved.
  Constants
   User
    Number: FIRST_ROW = 0
    Number: PREV_ROW  = 1
    Number: NEXT_ROW  = 2
    Number: LAST_ROW  = 3
    Number: ERROR_TIMEOUT = -1805
    Number: ERROR_ROWID = 806
  Resources
   ! Set the BMP images for push buttons as resources.
   Bitmap: resFirst
    File Name: FIRST.BMP
```

```
 Bitmap: resPrev
  File Name: PREV.BMP
 Bitmap: resNext
  File Name: NEXT.BMP
 Bitmap: resLast
  File Name: LAST.BMP
 Bitmap: resFetch
  File Name: FETCH.BMP
 Bitmap: resExit
  File Name: EXIT.BMP
 Bitmap: resOk
  File Name: OK.BMP
 Bitmap: resCancel
  File Name: CANCEL.BMP
 Bitmap: resInsert
  File Name: INSERT.BMP
 Bitmap: resDelete
  File Name: DELETE.BMP
 Bitmap: resUpdate
  File Name: UPDATE.BMP
 Bitmap: resUndo
  File Name: UNDO.BMP
Variables
 Sql Handle: hSqlSelect
 Sql Handle: hSqlUpdate
 Sql Handle: hSqlDelete
 Sql Handle: hSqlInsert
 Sql Handle: hSqlError
 Boolean: bConnectedSelect
 Boolean: bConnectedUpdate
 Boolean: bConnectedDelete
 Boolean: bConnectedInsert
 Boolean: bRollback
 String: strMessage
 Number: nError
 Number: nPos
Application Actions
 On SAM_AppStartup
  ! Login Dialog Box with defaults for database, user, password
  Set bConnectedSelect = SalModalDialog( dlgLogin, hWndNULL,
    'GUPTA', 'SYSADM', 'SYSADM', hSqlSelect )
  ! If one connect successful, set DBP_PRESERVE to TRUE and set
    the isolation level to Release Locks.
  If bConnectedSelect and
    SqlSetParameter( hSqlSelect, DBP_PRESERVE, TRUE, '') and
    SqlSetIsolationLevel( hSqlSelect, 'RL' )
```

```
    ! Make other necessary connect's.
    Set bConnectedUpdate = SqlConnect( hSqlUpdate )
    Set bConnectedDelete = SqlConnect( hSqlDelete )
    Set bConnectedInsert = SqlConnect( hSqlInsert )
    ! If successfully connected, create the main form.
    If bConnectedUpdate and bConnectedDelete and bConnectedInsert
     Call SalCreateWindow( frmMain, hWndNULL )
On SAM_SqlError
 ! Get the Sql Handle, error number and position of the last
   error.
 Call SqlExtractArgs( wParam, lParam, hSqlError, nError, nPos )
 ! See if the system initiated any rollback. hSqlError would be
   invalid if the error occurred during first SqlConnect.
   405 Invalid user name
   404 Invalid Password
   401 Cannot open database
 If nError != 405
   and nError != 404
   and nError != 401
   and hSqlError != hWndNULL
  Call SqlGetRollbackFlag( hSqlError, bRollback )
  If not bRollback
   ! Rollback the transaction so that we release any locks held
     before we display the dialog box.
   Call SqlImmediate ('ROLLBACK')
   Set bRollback = TRUE
  Set strMessage = 'The transaction has been rolledback. '
  Select Case nError
   Case ERROR_ROWID
    Set strMessage = strMessage ||
      'This record has been updated since you fetched it.
      Please Refresh and try again.'
    Break
   Case ERROR_TIMEOUT
    Set strMessage = strMessage ||
      'Timed out waiting for a lock. Please try again.
      You will have to Refresh after that.'
    Break
   Default
    Set strMessage = strMessage || 'Please Refresh.'
 Else
  Set strMessage = 'Enter the correct values and try again.'
 Call SalModalDialog
    ( dlgSqlError, hWndNULL, nError, strMessage )
 ! Return FALSE so that the Sql function which caused the error
   would return FALSE to its caller.
```

```
    Return FALSE
  On SAM_AppExit
    ! Time to disconnect all the connected Sql Handles.
    If bConnectedSelect
      Call SqlDisconnect( hSqlSelect )
    If bConnectedUpdate
      Call SqlDisconnect( hSqlUpdate )
    If bConnectedDelete
      Call SqlDisconnect( hSqlDelete )
    If bConnectedInsert
      Call SqlDisconnect( hSqlInsert )
```

Listing 3.1 Global variables and Application Actions for DATABASE.APP.

Defining Variables

The application defines some sql handles for connecting to the database. A Sql Handle is a data type to define or identify an open connection to a database. You declare a sql handle for each connection your application establishes with a database. Since I want one connection each for *selecting* all the records in COMPANY table, *updating* an existing record, *deleting* an existing record, and *inserting* a new record in the table, I have defined four such variables. In case of an error, hSqlError will store the error causing sql handle as you will see shortly in the discussion about On SAM_SqlError.

It is also important to remember which sql handles have been connected with the database successfully so that those sql handles can be disconnected when the application is terminated. Boolean variables such as bConnectedSelect serve this purpose. Boolean variable bRollback is used to indicate whether a transaction has been rolled back. Finally, I use nError and nPos to store the error number and the position within the SQL statement in case a SQL error occurs.

SAM_AppStartup

SAM_AppStartup message is sent to an application before any of the application's windows are created. You can process SAM_AppStartup and perform initialization tasks such as displaying a login dialog box that authorizes access to a database. This is the first message an application receives and it is sent only to the Application Actions section of the outline. In this application, I process this message and first call SalModalDialog to create the Login dialog box dlgLogin.

Displaying the Login Dialog Box

SQLWindows provides the SalModalDialog function to create a modal dialog box. A modal dialog box disables its owner window. In this case, since there is no window created so far, I have specified hWndNULL as the owner. This system variable represents a null window handle. In contrast to a modal dialog box, a system modal dialog box disables the entire Windows system.

SalModalDialog can pass information to the dialog box by accepting a variable number of parameters. The data types of these parameters must match the parameter data types of the dialog box being created. You define parameters for the dialog box in the Window Parameters section of the application outline. You can use window parameters to return information from the window being created by using a receive type.

In this application, I pass hSqlSelect as the receive parameter; if the user provides the correct user name and password for the database, the connection is successful and hSqlSelect is initialized. This is a good time to look at the code of the dialog box dlgLogin.

```
Dialog Box: dlgLogin
 Description: Dialog Box for login.
   It tries to connect one Sql Hanndle. If it succeeds,
   it sets the global system variables SqlDatabase, SqlUser and
   SqlPassword. Since one SqlConnect() was successful, the user
   name and password are verified to be correct. Other
   connections can be made using the same user name and password.
   The caller provides the default names to begin with.
 Contents
  Background Text: Database
  Background Text: User Name
  Background Text: Password
  Data Field: dfDatabase
  Data Field: dfUser
  Data Field: dfPassword
   Display Settings
    Format: Invisible
  Pushbutton: pbOk
   Keyboard Accelerator: Enter
   Message Actions
    On SAM_Click
     Set SqlDatabase = dfDatabase
     Set SqlUser = dfUser
```

```
      Set SqlPassword = dfPassword
      ! First connect could take a while, so show the hour glass
      Call SalWaitCursor( TRUE )
      If SqlConnect( hSqlLocal )
       Set hSqlParm = hSqlLocal
       ! Connect successful. Show the normal cursor again and end
         the dialog box.
       Call SalWaitCursor( FALSE )
       Call SalEndDialog( hWndForm, TRUE )
      ! Show the normal cursor again even though connect was not
        successful
      Call SalWaitCursor( FALSE )
     On SAM_Create
      ! Set the picture for the push button using the resource
      Call SalPicSet( hWndItem, resOk, PIC_FormatBitmap )
   Pushbutton: pbCancel
    Message Actions
     On SAM_Click
      ! The user has given up. End the dialog box.
      Call SalEndDialog( hWndForm, FALSE )
     On SAM_Create
      ! Set the picture for the push button using the resource
      Call SalPicSet( hWndItem, resCancel, PIC_FormatBitmap )
   Line
  Window Parameters
   ! Default database name, user name and password to begin with.
   String: strParmDatabase
   String: strParmUser
   String: strParmPassword
   Receive Sql Handle: hSqlParm
  Window Variables
   Sql Handle: hSqlLocal
  Message Actions
   On SAM_Create
    ! Use the default values supplied by the caller.
    Set dfDatabase = strParmDatabase
    Set dfUser = strParmUser
    Set dfPassword = strParmPassword
```

Listing 3.2 The Login dialog box – dlgLogin.

SAM_Create

SAM_Create is sent to a top-level window (dialog box, form window, or table window) and then to all of its children after they are created, but before they are

made visible. SAM_Create is also sent to an MDI window. After SQLWindows sends the SAM_Create messages, SQLWindows makes the form window and data fields visible. By processing the SAM_Create message, an application can perform initialization tasks. Typical initialization tasks include setting data field values and populating table windows and list boxes with data from a database.

You can see the use of SAM_Create in Listing 3.2. First, the dialog box itself gets the message, where I set the values of the data fields dfDatabase, dfUser, and dfPassword with the default values supplied in the parameters. You can also choose to set the values of these data fields in their own message actions section by processing SAM_Create.

Using Resources

Next in the order of events, push buttons pbOk and pbCancel get SAM_Create messages. As a good programming practice, you should never depend on which of the push buttons gets this message first. In response to SAM_Create, both these push buttons set the picture on them using the *resources* defined in the Resources section of the application outline as seen in Listing 3.1.

Resources let you specify bitmaps, icons, or cursors in the Global Declarations that you can refer to in the SalPicSet or SalCursorSet functions. Resources appear in the Outline Options dialog box if List Globals is checked. When you go into run mode at designtime, SQLWindows must be able to find the resources in external files. When you make an *.EXE or *.RUN version of the application, SQLWindows copies the resources from the external files into the application. You do not need to distribute the external files with a production version of an application.

You don't have to use resources; you can specify Picture Contents, File Name... through the customizer. Using resources saves space in the application if a resource is used more than once.

hWndItem

Notice the use of *hWndItem* while calling SalPicSet in Listing 3.1. hWndItem is a system variable—this variable is the window handle of the current object. When actions are executing from an object's Message Actions section, hWndItem is set to the window handle of that item. This can be a top level window or a child object.

Connecting to the Database

At this point, the dialog box just sits there and waits for the user to either change the values of the database, user name, or password or press one of the OK or Cancel push buttons. If the user presses the OK button, pbOk gets the SAM_Click message. Notice that pbOk has the Enter key defined as the Keyboard Accelerator, so even when the user uses the keyboard and presses the Enter key, pbOk receives the SAM_Click message. All the real work of the login dialog box is done here.

In response to this message, pbOk first copies the values of the data fields into system variables SqlDatabase, SqlUser, and SqlPassword. They are all string variables. SqlDatabase contains the name of the database to which the application connects. The default database name is DEMO. SqlUser contains a user name necessary to access a database. The default authorization name is SYSADM. Finally, SqlPassword contains a password necessary to access a database. The default password is SYSADM. Use these variables with the SqlConnect, SqlExists, and SqlImmediate functions.

The first connection to the database can take a while. Hence I use the SalWaitCursor function to display an hour glass prior to the first call to the SqlConnect function. Now, pbOk calls SqlConnect to connect with the database using a local sql handle. If everything goes well, the receive parameter hSqlParm is set to the local sql handle which is now connected. I use a local sql handle because SQLWindows does not permit passing a receive parameter to a window as a receive parameter to a function. The cursor changes back to normal again using the SalWaitCursor function and the dialog box is ended by calling SalEndDialog. The control returns to the caller of the SalModalDialog with a return value specified in the second parameter of SalEndDialog. Since everything went well, I return TRUE in this case.

SAM_SqlError

There is a possibility, however, that SQLWindows cannot open the specified database, the user name does not exist, or the password is not correct. When some SQL error occurs during SqlConnect or during any other SQL function, SQLWindows sends *SAM_SqlError* to the Application Actions section of the outline (see Listing 3.1). Here, you can control how the application responds to an error instead of using the default error processing which SQLWindows

provides. You can also use *When SqlError* in any actions section of the outline to process an error. I give you an example of When SqlError later in the book.

Upon receipt of the SAM_SqlError message, I first use SqlExtractArgs to get the sql handle causing the error, error number, and the position within the SQL statement where it occurred. SqlExtractArgs extracts this information from wParam and lParam. wParam and lParam are generic parameters used by SQLWindows Application Messages (SAM) to hold numeric information useful for message processing.

I check to see if the error is caused by an invalid database, user name, or password. Additionally, I check to see if the sql handle is not null because calling SqlGetRollbackFlag later with a null sql handle would result in an error.

SqlGetRollbackFlag

When a user wants to select, update, or delete a record while some other user still holds an exclusive lock on a record on the same page as this record, the first user cannot access the record and gets a time out waiting to acquire a shared or exclusive lock. To deal with such error conditions, I call SqlGetRollbackFlag to see if the transaction has been rolled back. SQLWindows sets the rollback flag when a system-initiated rollback occurs as the result of a deadlock or system failure. SQLWindows does not set the rollback flag on a user-initiated rollback. If there is no system-initiated rollback, I call SqlImmediate to prepare and execute a ROLLBACK statement.

If the transaction holds a lock, all other users connected to the same database may timeout also. It is very important to rollback the transaction before displaying any message to the user. The rollback ensures that the transaction does not hold any locks.

Hopefully, after some time, the transaction holding the exclusive lock commits and releases the lock.

Displaying Error Text, Reason, and Remedy

SalModalDialog is called to display the error message, error text, reason, and remedy. Listing 3.3 shows the dialog box dlgSqlError.

SqlErrorText function gets the error reason or remedy for the specified error code from ERROR.SQL. You can call SqlError to get the most recent error code. When your application detects an error condition, you can use the error code returned

by SqlError to look up the error reason and remedy with SqlErrorText. You can specify SQLERROR_Reason to retrieve the error code reason, SQLERROR_Remedy to retrieve the error message remedy, or use OR to get both reason and remedy.

SqlGetErrorTextX function returns the message text for a SQL error number from ERROR.SQL file.

Notice that for the multiline fields, I have specified 'Word Wrap' as Yes. If this customizer attribute is set to Yes, the text in the multiline field wraps. The default is no.

Coming back to Listing 3.1, after displaying the error, SAM_SqlError returns with a FALSE value. The function SqlConnect finally returns with the value FALSE. pbOk recognizes that SqlConnect did not succeed and simply displays the normal cursor back again.

```
Dialog Box: dlgSqlError
 Title: SQL Error
 Description: This dialog box displays the error text, cause and
   remedy corresponding to the SQL error nError.
 Contents
  Background Text: Message:
  Multiline Field: mlMessage
   Data
    Maximum Data Length: Default
    String Type: String
    Editable? No
   Display Settings
    Word Wrap? Yes
    Vertical Scroll? Yes
   Message Actions
    On SAM_Create
     Set mlMessage = strMessage
  Background Text: Reason:
  Multiline Field: mlCause
   Data
    Maximum Data Length: 512
    String Type: String
    Editable? No
   Display Settings
    Word Wrap? Yes
    Vertical Scroll? Yes
```

```
    Message Actions
     On SAM_Create
      Call SqlErrorText( nError, SQLERROR_Reason,
        mlCause, SalGetMaxDataLength( mlCause ), nMaxCause )
   Background Text: Remedy:
   Multiline Field: mlRemedy
    Data
     Maximum Data Length: 512
     String Type: String
     Editable? No
    Display Settings
     Word Wrap? Yes
     Vertical Scroll? Yes
    Message Actions
     On SAM_Create
      Call SqlErrorText( nError, SQLERROR_Remedy,
        mlRemedy, SalGetMaxDataLength( mlRemedy ), nMaxRemedy )
   Background Text: Error Text:
   Multiline Field: mlErrorText
    Data
     Maximum Data Length: 512
     String Type: String
     Editable? No
    Display Settings
     Word Wrap? Yes
     Vertical Scroll? Yes
    Message Actions
     On SAM_Create
      Set mlErrorText = SqlGetErrorTextX( nError )
   Pushbutton: pbDone
    Title: Done
    Message Actions
     On SAM_Click
      Call SalEndDialog( hWndForm, TRUE )
   Background Text: Error Number
   Data Field: dfError
    Message Actions
     On SAM_Create
      Set dfError = nError
  Window Parameters
   Number: nError
   String: strMessage
  Window Variables
   Number: nMaxCause
   Number: nMaxRemedy
```

```
Message Actions
  On SAM_Create
    ! Since we come here during the execution
      of a SQL operation, cursor might be a wait cursor.
      Show the normal cursor.
    Call SalWaitCursor( FALSE )
```

Listing 3.3 The dlgSqlError dialog box.

Ending the Dialog Box

At this point, the user can type in the correct values of the database name, user name, and password and press OK, or quit by pressing Cancel. If the user presses Cancel, pbCancel receives a SAM_Click message and in response to this message, ends the dialog box with FALSE as the second parameter to SalEndDialog. The control returns to the caller of SalModalDialog with a return value of FALSE so that it knows that the login did not succeed.

When the Login dialog box calls SalEndDialog, the control comes back to the SAM_AppStartup section of the application outline. If the login process succeeds, I set database parameter DBP_PRESERVE to TRUE and set *islolation level* to Release Locks.

DBP_PRESERVE—Cursor Context Preservation

In a SELECT statement (query), you identify tables, columns, and rows from which to select data. The database finds the data that matches the specification. In a *result set mode,* you can scroll through the results of such a query. SQLBase uses a cache on the server side that holds the results of a query to support browsing. However, other databases such as Sybase SQL Server do not support this feature. Instead, SQLWindows connectivity maintains a cache on the client machine. This feature is called *frontend result sets.*

However, if a new record is inserted by this user or some other user connected to the same database, it does not appear in the result set. Similarly, if a record is deleted, it continues to appear in the result set. Of course, whenever this application tries to fetch this record directly from the server, the server indicates that the record has been deleted. As you will see later, I provide a Refresh push button on the toolbar of the main form. Pressing this push button prepares and executes the SELECT statement, creating the result set again.

SQLWindows applications default to result set mode.

You can use DBP_PRESERVE to get or set the value of the database parameter that specifies whether cursor context preservation is on or off. If cursor-context preservation is on, a COMMIT does not destroy an active result set (cursor context). This enables an application to maintain its position after a COMMIT, INSERT, or UPDATE.

The cursor context is not preserved after an isolation level change. The context is preserved after a ROLLBACK if both of the following are true:

- The application is in the Release Locks (RL) isolation level.
- A DDL (Data Definition Language) operation was not performed.

If cursor-context preservation is off (FALSE), a COMMIT does destroy an active result set. Cursor context preservation is lost. You can use this constant with the SqlGetParameter and SqlSetParameter functions.

In this application, I set DBP_PRESERVE to TRUE because I do not want the result set to be destroyed when I COMMIT after updating a record, deleting a record, or inserting a new record. If I did not set it to TRUE, after a COMMIT operation, the result set generated by the SELECT statement for browsing all the records would be destroyed and it would be necessary to prepare and execute the SELECT statement again. This results in unnecessary delay each time a record is updated, deleted, or inserted. Also, the user would be annoyed at having to go back to the first record of the result set each time such a modification occurs.

Isolation Levels

Isolation refers to the extent to which operations performed by one user can be affected by operations performed by another user. Isolation levels control the extent to which changes made by one user affect another user accessing the same table or tables. SQLBase supports the following isolation levels:

- *CS—Cursor Stability.* This isolation level locks only the page that you are currently processing from other users. SQLBase places a shared lock on a page for as long as the cursor is on that page. It holds exclusive locks and shared locks until a COMMIT. Other pages you access during the transaction become available to other users and they do not have to wait for

your COMMIT. Data that you read during a transaction can be changed by other users once your cursor moves on to a new page. SQLBase sends only one row to the input message buffer despite the size of the buffer. In other words, each SqlFetchNext or SqlFetchPrevious causes the client and server to exchange messages across the network.

Use Cursor Stability when you want to update one row at a time using the CURRENT OF <cursor> clause. When SQLWindows fetches the row into the client's input message buffer, the page has a shared lock which means that no other transaction can update it.

- *RL—Release Locks.* This isolation level increases concurrency by releasing all shared locks by the time control returns to the client. In contrast, under Cursor Stability, when you move off a database page, SQLBase drops the shared lock acquired when you read the page; however, if a row from the page is still in your input message buffer, the page remains locked. This isolation level fills the input message buffer with rows, minimizing network traffic. You should use this isolation level for browsing applications which display a set of rows to the user.

- *RO—Read Only.* This isolation level places no locks on the database and you can only use it for reading data. SQLWindows does not let you execute Data Definition Language (DDL) and Data Manipulation Language (DML) statements while a transaction is in Read Only mode. This isolation level provides a view of the data as it existed when the transaction began. If you request a page that is currently locked by another transaction, SQLBase provides an older copy of the page from the read-only history file. The read-only history file maintains multiple copies of database pages that have changed.

  This is an appropriate isolation level if you want the data you read to be consistent, but you do not necessarily need it to be current. This isolation level also guarantees maximum concurrency. Read-only transactions may affect performance, so SQLBase disables them by default. SQLBase allows read-only transactions only when the readonly=1 statement is specified in the SQL.INI file, or when you issue a SET READONLY 1 command, or when an application issues a call to sqlset using the SQLPROD parameter. This isolation level fills the input message buffer with rows.

- *RR—Read Repeatability*. This isolation level locks all pages that you access from other users until you COMMIT your transaction. If you re-read a page during a transaction, you see the same data. Read Repeatability guarantees that the data you access is consistent for the life of a transaction. Read Repeatability is the default SQLBase isolation level. This isolation level fills the input message buffer with rows. All shared locks remain regardless of the size of the input message buffer until the application issues a COMMIT or ROLLBACK statement.

Unless you use named transactions as I explain later, the scope of a transaction is *all connections* that an application makes to a given database and user name. You choose an isolation level based on the application's requirements for consistency and concurrency. An isolation level applies to *all* sql handles in a transaction. Changing the isolation level causes an implicit commit on all sql handles in that transaction. However, calling SqlSetIsolationLevel and specifying the isolation level already set does not cause an implicit commit.

In this application, I use Release Locks (RL) instead of the SQLBase default, Read Repeatability (RR). My main concern while writing this application was to increase concurrency so that several users can browse and update the records. Since all the shared locks are released by the time the control returns to the client, even many such readers do not interfere with someone who is trying to update a record. If the shared locks were still held by the readers, an update operation would fail to acquire an exclusive lock. This would mean that no one would be able to update a record as long as there was even one reader trying to read from the same page.

Coming back to Listing 3.1, if everything goes well with setting DBP_PRESERVE, and the isolation level, I connect other necessary sql handles for the insert, update and delete operations using the system variables SqlDatabase, SqlUser, and SqlPassword set earlier by the login dialog box. If there are no problems encountered, I finally create the main window by calling SalCreateWindow. SalCreateWindow creates modeless dialog boxes, MDI windows, form windows, and top-level table windows at runtime. If you specify an owner (hWndOwner—the second parameter), the new window always displays on top of its owner, closes when its owner closes, and hides when its owner is minimized. SalCreateWindow can pass data to and from the window being created by accepting a variable number of parameters. I will shortly describe what happens

when this form is created. In Listing 3.4, note that I have specified the 'Automatically Created at Runtime?' setting to be No.

If the login process did not succeed, I do not do anything and SQLWindows sends a SAM_AppExit message to the application. The SAM_AppExit message is also sent after the main window is destroyed. At this time, I check to see if each sql handle is connected or not. If it is connected, I disconnect the sql handle from the database using the SqlDisconnect function. Disconnecting the last Sql Handle from a database causes an implicit COMMIT of the database.

The Main Form Window frmMain

In this section, I describe the main form frmMain. Most of the work is really done by this form. frmMain creates the result set using a SELECT statement, provides push buttons on the toolbar to go to the first, previous, next, and last records, and provides push buttons to update and delete an existing record, and to insert a new record.

Keeping Track of Changes

The user can make changes to any record and commit the changes by pressing the Update push button. In order to prevent the user from losing any changes made without first updating the database, I use bFormDirty to keep track of whether any changes have been made to the current record. All the data fields on the form such as dfName, dfAddr1, etc., help maintain this boolean variable bFormDirty. I use a SAM_Validate message to help me with this.

SAM_Validate

SAM_Validate is sent to a Data field, multiline field, combo box, and column when the user changes the value of the object and then moves the focus away from the object. The user can move the focus by several actions such as tabbing to another object, clicking another object, or using a mnemonic or accelerator to activate another object. When a user changes one of these objects, the object's field edit flag changes to TRUE. You can get and set the field edit flag using SalQueryFieldEdit and SalSetFieldEdit.

At this point, I would like to mention that SAM_Validate is normally used to validate the value of the object. For example, if you have a data field for interest

rate, you may want to make sure that it is between 0 and 100. In this application, I am using SAM_Validate to simply set the value of bFormDirty.

SAM_Validate or SAM_AnyEdit?

Alternately, I could use SAM_AnyEdit but that would set bFormDirty everytime a key is pressed to change a data field—quite unnecessary when we have SAM_Validate which is sent only when user is moving the focus away from this object and the value of the object has changed. In Chapter 5, I use SAM_AnyEdit instead of SAM_Validate. This is because while using the application developed in Chapter 5, a user can choose a *menu item* to leave the current record, and selecting a menu item from the menu does not send a SAM_Validate message.

bFormDirty is reset each time information about a new record is brought into the data fields.

When the form is created, it receives SAM_Create and in response, it sets the values of certain strings for containing the actual SQL statements. We will look at these SQL statements later in the discussion about the push buttons on the toolbar. It resets the value of bFormDirty and posts a SAM_Click message to the push button pbRefresh of the toolbar.

SalPostMsg or SalSendMsg?

SalPostMsg *posts* the specified message to a window by adding the specified message to the receiver's message queue. SalPostMsg returns immediately and control returns to the statement after the call to the SalPostMsg function. There is another function SalSendMsg which *sends* the specified message to a window. SalSendMsg does not return until the processing for the message is complete.

```
Form Window: frmMain
 Title: Maintain Records of COMPANY Table
 Display Settings
  Automatically Created at Runtime? No
 Description: The main form to maintain GUPTA.Company table.
 Menu
  Popup Menu: &File
   Menu Item: E&xit
    Status Text: Exit this application
    Menu Actions
     Call SalPostMsg( hWndForm, SAM_Close, 0, 0 )
   Menu Item: A&bout...
    Status Text: Display the About Dialog Box
```

```
    Menu Actions
     Call SalModalDialog( dlgAbout, hWndForm )
 Toolbar
  Contents
   Pushbutton: pbFirst
   Pushbutton: pbPrev
   Pushbutton: pbNext
   Pushbutton: pbLast
   Pushbutton: pbRefresh
   Pushbutton: pbInsert
   Pushbutton: pbDelete
   Pushbutton: pbUpdate
   Pushbutton: pbUndo
   Pushbutton: pbExit
 Contents
  Background Text: Id
  Background Text: Name
  Background Text: Phone
  Background Text: Fax
  Background Text: Addr1
  Background Text: Addr2
  Background Text: City
  Background Text: State
  Background Text: ZIP
  Background Text: Country
  Data Field: dfId
   Data
    Data Type: Number
    Editable? No
  Data Field: dfName
   Message Actions
    On SAM_Validate
     Set bFormDirty = TRUE
  Data Field: dfPhone
   Message Actions
    On SAM_Validate
     Set bFormDirty = TRUE
  Data Field: dfFax
   Message Actions
    On SAM_Validate
     Set bFormDirty = TRUE
  Data Field: dfAddr1
   Message Actions
    On SAM_Validate
     Set bFormDirty = TRUE
```

```
Data Field: dfAddr2
 Message Actions
  On SAM_Validate
   Set bFormDirty = TRUE
Data Field: dfCity
 Message Actions
  On SAM_Validate
   Set bFormDirty = TRUE
Data Field: dfState
 Message Actions
  On SAM_Validate
   Set bFormDirty = TRUE
Data Field: dfZIP
 Message Actions
  On SAM_Validate
   Set bFormDirty = TRUE
Data Field: dfCountry
 Message Actions
  On SAM_Validate
   Set bFormDirty = TRUE
Data Field: dfRowid
 Display Settings
  Visible? No
Functions
 Function: OkToLoseChangesIfAny
  Description: See if the form is dirty (changes made to any
    data fields). If yes, ask user if it's ok to lose changes.
  Returns
   Boolean:
  Actions
   If bFormDirty and
     SalMessageBox( 'Lose Changes?', 'Confirmation',
     MB_YesNo | MB_IconQuestion | MB_DefButton2) = IDNO
    Return FALSE
   Else
    Set bFormDirty = FALSE
    Return TRUE
 Function: MessageBoxIfFetchError
  Description: This function displays a message box
    in case First, Next, Previous, Last return FETCH_EOF,
    FETCH_Delete, or FETCH_Update.
  Returns
   Boolean:
  Parameters
   Number: nInd
```

```
    Local variables
     String: strMessage
    Actions
     Select Case nInd
      Case FETCH_EOF
       Set strMessage = 'Reached the end.'
       Break
      Case FETCH_Update
       Set strMessage = 'The record has been updated.'
       Break
      Case FETCH_Delete
       Set strMessage = 'The record has been deleted.'
       Break
      Default
       Return TRUE
     Call SalMessageBeep( 0 )
     Call SalMessageBox( strMessage, 'FETCH Information', MB_Ok )
     Return FALSE
Window Variables
 String: strSelect
 String: strUpdate
 String: strDelete
 String: strInsert
 String: strUpdateId
 String: strSelectId
 Number: nFetchResult
 Boolean: bOk
 Number: nRows
 Number: nRowNumber
 Boolean: bFormDirty
 Window Handle: hWndDatafield
Message Actions
 On SAM_Create
  ! Set the strings for SELECT, UPDATE, INSERT and DELETE
    statements. Note the use of ROWID.
  Set strSelect = 'SELECT ID, NAME, PHONE, FAX,
    ADDR1, ADDR2, CITY, STATE, ZIP, COUNTRY, ROWID
    FROM COMPANY
    INTO :dfId, :dfName, :dfPhone, :dfFax, :dfAddr1, :dfAddr2,
    :dfCity, :dfState, :dfZIP, :dfCountry, :dfRowid
    ORDER BY ID'
  ! dfId is a read only data field, so it doesn't need to be
    updated.
  Set strUpdate = 'UPDATE COMPANY
    SET NAME=:dfName, PHONE=:dfPhone, FAX=:dfFax,
    ADDR1=:dfAddr1, ADDR2=:dfAddr2, CITY=:dfCity,
```

```
    STATE=:dfState, ZIP=:dfZIP, COUNTRY=:dfCountry
    WHERE ROWID=:dfRowid'
  Set strInsert = 'INSERT INTO COMPANY
    (ID, NAME, PHONE, FAX, ADDR1, ADDR2, CITY, STATE, ZIP,
    COUNTRY)
    VALUES (:dfId, :dfName, :dfPhone, :dfFax, :dfAddr1,
    :dfAddr2, :dfCity, :dfState, :dfZIP, :dfCountry)'
  Set strDelete = 'DELETE FROM COMPANY WHERE ROWID=:dfRowid'
  ! Increment the ID field of COMPANY_ID
  Set strUpdateId = 'UPDATE COMPANY_ID SET ID = ID + 1'
  ! Get the old value of ID field
  Set strSelectId = 'SELECT (ID - 1) INTO :dfId FROM COMPANY_ID'
  ! Mark the form as not dirty (no changes yet)
  Set bFormDirty = FALSE
  ! Refresh - create the result set
  Call SalPostMsg( pbRefresh, SAM_Click, 0, 0 )
 On SAM_Close
  If not OkToLoseChangesIfAny(  )
   ! Return FALSE only if the form is dirty and the user doesn't
     want to lose changes. Otherwise, it's ok to leave.
   Return FALSE
```

Listing 3.4 The main form window frmMain.

SAM_Close

When the user presses Exit push button on the toolbar, chooses Exit from the File menu, Close from the system menu, or the message is sent programmatically in the case of an error, the form receives a SAM_Close message. SAM_Close is sent to a dialog box, a form window, top-level QuestWindow, MDI window, or a top-level table window when you choose the Close command from a window's system menu or when you double-click on the window's system menu.

SAM_Close is sent to an application to notify the application that a user is attempting to close a window. By processing the SAM_Close message, an application can check if there is data that needs to be saved. A SAM_Close message is not sent on a call to SalQuit, SalEndDialog, or SalDestroyWindow.

If SAM_Close is sent to the Message Actions section of a modal or system modal dialog box, the application can call SalMessageBox to prevent the destruction of the dialog box. Otherwise, default processing closes the message box.

If the window processing SAM_Close does not execute a return, the window closes; if SAM_Close returns FALSE (or TRUE), the window does *not* close.

In this application, upon receiving SAM_Close, the form window checks to see if the user has made any changes to the record. If yes, it asks the user if it's OK to lose changes. This is done by the function OkToLoseChangesIfAny by calling SalMessageBox.

Notice the use of MB_YesNo | MB_IconQuestion | MB_DefButton2. This way, the message box has two buttons, Yes and No. It displays a question mark icon, and the default button is the second one—No. If the user chooses No from this message box, FALSE is returned from the On SAM_Close section indicating the window should not be closed. In all other cases, there is no return and the window closes.

Let's now see what each of the push buttons on the toolbar do.

Creating Result Set

The main purpose of pbRefresh is to prepare and execute the SELECT statement to create the result set, and to post a SAM_Click message to pbFirst so that the first record is displayed. Let us see the code that does that.

```
Pushbutton: pbRefresh
 Title: Refresh
 Message Actions
  On SAM_Click
   If OkToLoseChangesIfAny( )
    ! Refreshing may take a while, so wait cursor
    Call SalWaitCursor( TRUE )
    If not SqlPrepareAndExecute( hSqlSelect, strSelect )
     Call SalMessageBox(
       'Could not prepare or execute the SELECT statement.
       Exiting.', 'Serious Error', MB_Ok | MB_IconStop )
     ! Time to quit. Use SalSendMsg instead of SalPostMsg
     Call SalSendMsg( hWndForm, SAM_Close, 0, 0 )
    Else
     ! Execute was successful; now fetch the first record
     Call SalPostMsg( pbFirst, SAM_Click, 0, 0 )
     ! Mark bRollback as FALSE as the SELECT was just
       executed.
     Set bRollback = FALSE
    ! Don't forget to get the normal cursor back
```

```
        Call SalWaitCursor( FALSE )
    On SAM_Create
        ! Set the picture for the push button using the resource
        Call SalPicSet( hWndItem, resFetch, PIC_FormatBitmap )
```

Listing 3.5 The pbRefresh push button.

Preparing (Compiling) a SQL Statement

Before you can execute a SQL statement, you must prepare (compile) it. Function SqlPrepare compiles a SQL statement. Compiling includes:

- Checking the syntax of the SQL statement.
- Checking the system catalog.
- Processing a SELECT statement's INTO clause. An INTO clause names where data is placed when it is fetched. These variables are sometimes called *into* variables. You can specify up to 255 into variables per SQL statement.
- Identifying *bind* variables in the SQL statement. Bind variables contain input data for the statement. Bind variables are used within the WHERE clause of a SQL statement, the VALUES clause of an INSERT statement, or the SET clause of an UPDATE statement. You can specify up to 255 bind variables per SQL statement.

SqlPrepare is optionally followed by a SqlOpen, SqlExecute, SalTblDoInserts, SalTblDoUpdates, SalTblDoDeletes, or fetches.

Executing a SQL Statement

SqlExecute executes a SQL statement that was prepared with SqlPrepare or retrieved with SqlRetrieve. Bind variables are sent to the database when you call SqlExecute. SqlExecute does not fetch data. To fetch data, you should call one of the SqlFetch* functions: SqlFetchNext, SqlFetchPrevious, or SqlFetchRow. In this application, pbRefresh posts a SAM_Click message to pbFirst which, as we will see shortly, calls SqlFetchRow.

You can combine both prepare and execute by calling one function SqlPrepareAndExecute.

Let me reproduce here the SELECT statement from Listing 3.4. strSelect is set to:

```
'SELECT ID, NAME, PHONE, FAX, ADDR1, ADDR2, CITY, STATE,
ZIP, COUNTRY, ROWID FROM COMPANY INTO :dfId, :dfName,
:dfPhone, :dfFax, :dfAddr1, :dfAddr2, :dfCity, :dfState,
:dfZIP, :dfCountry, :dfRowid ORDER BY ID'.
```

The INTO clause contains INTO variables such as :dfId, :dfName, etc. These are the names of the data fields on the form. dfRowid is a hidden field and is used to store the ROWID.

ROWID

ROWID is a feature of SQLBase which enables you to identify not only a row but a version of a row in a table. ROWID is used to avoid overwriting another user's update in a multi-user environment. SQLBase itself assigns each row a unique string identifier when a new row is inserted in a table. SQLBase also changes this identifier automatically whenever a user updates it. I will show you how I avoid overwriting another user's updates when I discuss pbDelete and pbUpdate.

Other databases may have similar features to avoid overwriting other users' modifications in a multi-user environment.

Finally, it resets a flag bRollback. I use this flag to indicate that the transaction has been rolled back.

Fetching First Row

The main purpose of pbFirst is to fetch the first row of the result set. It first makes sure that either no data fields on the form have changed or it is OK with the user to lose the changes. Since the row numbers are 0-based, it calls SqlFetchRow with 0 as the row number. nFetchResult contains the result of this operation. It can contain one of the following values:

Constant	Description
FETCH_Ok	This constant is returned to one of the SqlFetch* functions to indicate that the requested row was successfully fetched from the query set.
FETCH_Update	Indicates that the row has been updated since the time the SELECT statement was executed.

Constant	Description
FETCH_Delete	Indicates a failure in fetching the requested row. SQLWindows could not fetch the row because it was deleted since the time the SELECT statement was executed. The result set remains active. All the INTO variables contain NULL values.
FETCH_EOF	Indicates a failure in fetching the requested row. SQLWindows could not fetch the row because it reached the end of the result set.

You can use these constants with the SqlFetchNext, SqlFetchPrevious, and SqlFetchRow functions.

```
Pushbutton: pbFirst
 Title: First
 Message Actions
  On SAM_Click
   If OkToLoseChangesIfAny(  )
    ! Fetch the first row : row number 0.
    Set bOk = SqlFetchRow( hSqlSelect, 0, nFetchResult )
    If bOk
     Call MessageBoxIfFetchError( nFetchResult )
     ! Remember where we are in the result set
     Set nRowNumber = 0
     ! Enable Next and Last. Disable First and Previous
     Call SalEnableWindow( pbNext )
     Call SalEnableWindow( pbLast )
     Call SalDisableWindow( pbFirst )
     Call SalDisableWindow( pbPrev )
  On SAM_Create
   ! Set the picture for the push button using the resource
   Call SalPicSet( hWndItem, resFirst, PIC_FormatBitmap )
```

Listing 3.6 The pbFirst push button.

Function MessageBoxIfFetchError is called to display an appropriate message if nFetchResult is not FETCH_Ok. If everything goes well, pbFirst disables pbFirst and pbPrev, and enables pbNext and pbLast. Since I also want to keep track of the row number being displayed on the form, I reset nRowNumber.

Fetching Next Row

pbNext calls SqlFetchNext to fetch the next row. Here are the relevant portions of code:

```
Pushbutton: pbNext
 Title: Next
 Message Actions
  On SAM_Click
   If OkToLoseChangesIfAny(  )
    ! Fetch the next row of the result set.
    Set bOk = SqlFetchNext( hSqlSelect, nFetchResult )
    If bOk
     Call MessageBoxIfFetchError( nFetchResult )
     ! Remember where we are in the result set
     Set nRowNumber = nRowNumber + 1
     Call SalEnableWindow( pbFirst )
     Call SalEnableWindow( pbPrev )
    Else
     ! Could not go to the next row (last row?).
     If nFetchResult = FETCH_EOF
      Call SalDisableWindow( pbNext )
      Call SalDisableWindow( pbLast )
  On SAM_Create
   ! Set the picture for the push button using the resource
   Call SalPicSet( hWndItem, resNext, PIC_FormatBitmap )
```

Listing 3.7 The pbNext push button.

Fetching Previous Row

pbPrev does pretty much the same as pbNext except that it calls SqlFetchPrevious to fetch the previous row. Here are the relevant portions of code:

```
Pushbutton: pbPrev
 Title: Prev
 Message Actions
  On SAM_Click
   If OkToLoseChangesIfAny(  )
    ! Fetch the previous row of the result set.
    Set bOk = SqlFetchPrevious( hSqlSelect, nFetchResult )
    If bOk
     Call MessageBoxIfFetchError( nFetchResult )
     ! Remember where we are in the result set
```

```
        Set nRowNumber = nRowNumber - 1
        ! Enable Next and Last
        Call SalEnableWindow( pbLast )
        Call SalEnableWindow( pbNext )
       Else
        ! Could not go to the previous row (first row?).
        If nFetchResult = FETCH_EOF
         Call SalDisableWindow( pbFirst )
         Call SalDisableWindow( pbPrev )
    On SAM_Create
     ! Set the picture for the push button using the resource
     Call SalPicSet( hWndItem, resPrev, PIC_FormatBitmap )
```

Listing 3.8 The push button pbPrev.

Fetching the Last Row

To go to the last row, pbLast first calls SqlGetResultSetCount to get the number of rows in the result set. This function counts the rows in a result set by building

```
 Pushbutton: pbLast
  Title: Last
  Message Actions
   On SAM_Click
    If OkToLoseChangesIfAny(  )
     ! First see how many rows there are in the result set
     If SqlGetResultSetCount( hSqlSelect, nRows )
      ! Fetch the last row : row number nRows-1 because it's 0-
        based
      Set bOk =
        SqlFetchRow( hSqlSelect, nRows-1, nFetchResult )
      If bOk
       Call MessageBoxIfFetchError( nFetchResult )
       ! Remember where we are in the result set (last row, 0-
         based)
       Set nRowNumber = nRows - 1
       ! Disable Next and Last. Enable First and Previous
       Call SalEnableWindow( pbFirst )
       Call SalEnableWindow( pbPrev )
       Call SalDisableWindow( pbNext )
       Call SalDisableWindow( pbLast )
```

```
On SAM_Create
  ! Set the picture for the push button using the resource
  Call SalPicSet( hWndItem, resLast, PIC_FormatBitmap )
```

Listing 3.9 The pbLast push button.

the result set. SQLWindows fetches each row that has not already been fetched, returns a count of the rows, and positions the cursor back to its original position. This can be time-consuming if the result set is large. You must be in Result Set mode. You must call SqlExecute before SqlGetResultSetCount. Once it knows the total number of rows in the result set, it calls SqlFetchRow to fetch the last row.

Deleting a Record

There are two possibilities when the user presses the Delete push button. There is a possibility that the user had earlier pressed the Insert push button and was just typing some values in the data fields. In this case, there is no real row to be deleted from the database. Simply posting a SAM_Click message to pbUndo would bring back the row the user was viewing before pressing the Insert push button. The other possibility is that the user really wanted to delete an existing row from the table. For this, pbDelete needs to prepare and execute a Delete statement. To find out whether the user was inserting a new record or was viewing an existing record, I examine the dfId data field of the form. As you will see shortly, dfId is blank when a user presses the Insert push button but has not pressed the Update push button yet. For an existing record, this data field will always contain a value.

Deleting a Record in a Multi-user Environment

Let us revisit the DELETE statement as set in Listing 3.4. strDelete was set to 'DELETE FROM COMPANY WHERE ROWID=:dfRowid'. We could have used ID=:dfId in the where clause instead, but that would not be appropriate in a multi-user environment. Let me explain why. Let us consider the following scenario in which two users USER1 and USER2 are both running this application at the same time.

USER1	USER2
1. Fetches row with ID=10.	1. Fetches row with ID=10

USER1	USER2
2. Updates row with ID=10. Since USER2 has released the shared lock, USER1 is able to acquire exclusive lock and also commit the transaction.	
	2. Looks at the stale information for ID=10 and decides to delete it. By now, USER1 has committed, so there are no locks on the record. USER2 is able to acquire exclusive lock and delete the record. USER2 just deleted the record based on the stale information! If USER2 knew the latest changes by USER1, maybe USER2 would have decided not to delete the record.

Let me show you how using ROWID would have saved USER2 from making this mistake.

USER1	USER2
1. Fetches row with ID=10. The ROWID, say, ROWID1 is also fetched.	1. Fetches row with ID=10. The ROWID (ROWID1) is also fetched.
2. Updates row with ROWID = ROWID1. Since USER2 has released the shared lock, USER1 is able to acquire exclusive lock and also commit the transaction. But since the row has been modified, SQLBase changes the ROWID to, say, ROWID2.	

USER1	USER2
	2. Looks at the stale information for ID=10 and ROWID = ROWID1 and decides to delete it. It issues a DELETE statement to delete the row with ROWID = ROWID1. But there is no such row in the table and the delete operation fails!

At this point, the USER2 has the option of looking at the latest value of the row and decide based on the latest information whether to delete it or not. That is precisely what pbDelete does. It marks the form as dirty and posts a SAM_Click message to pbRefresh. If the user chooses, the result set will be created again fetching the latest information.

If everything goes well, a SAM_Click message is sent to pbNext if the row just deleted was the first one or a SAM_Click message is sent to pbPrev if the row was not the first one. If the transaction has been rolled back, nothing is done because the result set may be destroyed.

```
Pushbutton: pbDelete
 Title: Delete
 Message Actions
  On SAM_Click
   ! Get the confirmation for the delete operation
   If SalMessageBox( 'Are you sure?', 'Confirmation',
     MB_YesNo | MB_IconQuestion | MB_DefButton2) = IDYES
    ! See if we are in the middle of insert operation
      If dfIs is null, it means we were inserting a new
      record.
    If not SalIsNull( dfId )
     ! Not in the middle of insert operation
     Call SalWaitCursor( TRUE )
     If not SqlPrepare( hSqlDelete, strDelete )
      Call SalMessageBox( 'Could not prepare the Delete
      statement. Exiting.',
      'Serious Error', MB_Ok | MB_IconStop )
      ! Time to quit
      Call SalSendMsg( hWndForm, SAM_Close, 0, 0 )
     If SqlExecute( hSqlDelete )
```

```
        ! Executed the DELETE statement. Now COMMIT it.
        Call SqlCommit( hSqlDelete )
        ! Mark the form as not dirty (no changes yet)
        Set bFormDirty = FALSE
        ! Current record is deleted. Display the previous one.
          If the deleted record was the first one, display the
          next one. If the transaction has been rolledback,
          don't do anything.
        If not bRollback
         If nRowNumber > 0
          Call SalPostMsg( pbPrev, SAM_Click, 0, 0 )
         Else
          Call SalPostMsg( pbNext, SAM_Click, 0, 0 )
       Call SalWaitCursor( FALSE )
      Else
       ! In the middle of insert operation. The record is not
         INSERTed yet, so simply undo all the changes to this
         record.
       Call SalPostMsg( pbUndo, SAM_Click, 0,0)
    On SAM_Create
     ! Set the picture for the push button using the resource
     Call SalPicSet( hWndItem, resDelete, PIC_FormatBitmap )
```

Listing 3.10 The pbDelete push button.

Inserting a New Record

You might be surprised, but pbInsert does not do a lot. It just empties all the fields on the form. When the user has typed in all the information except ID, which is a read-only data field, the user presses the Update push button which is where the real insert operation, including obtaining a new ID number, is done.

```
Pushbutton: pbInsert
 Title: Insert
 Message Actions
  On SAM_Click
   If OkToLoseChangesIfAny(  )
    ! Clear all the data fields on the form. Get the first
      data field on hWndForm.
    Set hWndDatafield = SalGetFirstChild( hWndForm,
        TYPE_DataField )
    While hWndDatafield != hWndNULL
     ! Clear the data field
     Call SalClearField( hWndDatafield )
     ! Now get the next data field. Note the use of
```

```
        hWndDatafield; not hWndForm for the first parameter.
      Set hWndDatafield = SalGetNextChild( hWndDatafield,
          TYPE_DataField )
   On SAM_Create
    ! Set the picture for the push button using the resource
    Call SalPicSet( hWndItem, resInsert, PIC_FormatBitmap )
```

Listing 3.11 The push button pbInsert.

Finding All Children

To empty all the data fields on the form, I use three functions: SalGetFirstChild, SalGetNextChild and SalClearField.

SalGetFirstChild

SalGetFirstChild returns the handle of the first child window of the specified type. For example, if you were interested only in data fields, push buttons, and radio buttons, you specify TYPE_DataField | TYPE_PushButton | TYPE_RadioButton. Since I am only interested in emptying the data fields, I specify just TYPE_DataField.

SalGetNextChild

SalGetFirstChild gets the window handle of the first child of the specified type, while I use SalGetNextChild in a loop to get the next child. This function should really be called SalGetSibling because it takes the window handle of the child window and not the parent window as the first parameter.

SalClearField

For each of these child objects (data fields), I use SalClearField to make them blank. This function clears the value from a data field, multiline field, or table window column.

When there are no more children left, SalGetFirstChild and SalGetNextChild return hWndNULL and the While loop ends.

Committing Changes

This is the place where the real insert or update operation takes place. An INSERT operation needs to take place if the user wants to insert a new row in the table. An UPDATE operation is needed if the user wants to modify an existing

row. Like pbDelete, pbUpdate also finds out which of the two operations to perform by examining the data field dfId.

```
Pushbutton: pbUpdate
 Title: Update
 Message Actions
  On SAM_Click
   Call SalWaitCursor( TRUE )
   ! See if we are in the middle of insert operation. If dfIs
     is null, it means we were inserting a new record.
   If SalIsNull( dfId )
    ! Insert operation
    ! Increment the ID number in COMPANY_ID table and select
      the old value. To generate the next sequential ID.
      Updating puts the exclusive lock so no one else would
      be able to read it until we COMMIT.
    If (SqlImmediate(strUpdateId)
      and SqlImmediate(strSelectId)
      and SqlPrepareAndExecute( hSqlInsert, strInsert ))
     ! Time to commit the transaction. Note that it will
       commit both the changes to the record as well as the ID
       in the COMPANY_ID table.
     Call SqlCommit( hSqlInsert )
     ! Mark the form as not dirty (no changes yet)
     Set bFormDirty = FALSE
     ! Now re-fetch the current row of the result set.
     ! In case, the UPDATE failed last time and rollback
       occurred, don't fetch the row as the SELECT may not
       have survived the rollback.
     If not bRollback
      Call SqlFetchRow( hSqlSelect, nRowNumber, nFetchResult )
   Else
    ! Normal Update - not insert operation
    If not SqlPrepare( hSqlUpdate, strUpdate )
     Call SalMessageBox(
       'Could not prepare the Update statement',
       'Serious Error', MB_Ok | MB_IconStop )
     ! Time to quit
     Call SalSendMsg( hWndForm, SAM_Close, 0, 0 )
    If SqlExecute( hSqlUpdate )
     ! UPDATE was successful, now COMMIT it.
     Call SqlCommit( hSqlUpdate )
     ! Mark the form as not dirty (no changes yet)
     Set bFormDirty = FALSE
     ! Now re-fetch the same row of the result set. One reason
```

```
        we have been keeping track of nRowNumber. This
        refetching is necessary to get the new ROWID so that if
        the user makes a change to this record again without
        first refreshing, invalid ROWID error would not occur.
        In case, the UPDATE failed last time and rollback
        occurred, don't fetch the row as the SELECT would not
        have survived the rollback.
      If not bRollback
       Call SqlFetchRow( hSqlSelect, nRowNumber, nFetchResult )
    Call SalWaitCursor( FALSE )
   On SAM_Create
    ! Set the picture for the push button using the resource
    Call SalPicSet( hWndItem, resUpdate, PIC_FormatBitmap )
```

Listing 3.12 The pbUpdate push button.

Inserting a New Record

In this case, the user must have earlier pressed the INSERT push button and, possibly, entered some values in the data fields of the form. Since dfID data field is read-only, it is blank at the moment. When the user presses this push button, pbUpdate has to calculate the next ID to use for this new record, increment the ID for future use, insert the record with the ID just calculated, and commit the transaction.

Calculating Next Sequential Number

To remember the next ID to use, I make use of the COMPANY_ID table, which contains just one row and one column. This is the next ID to use. I first update this row using the following SQL statement:

```
'UPDATE COMPANY_ID SET ID = ID + 1'
```

After having done this, I get the old value of ID by executing the following SQL statement:

```
'SELECT (ID - 1) INTO :dfId FROM COMPANY_ID'
```

I execute UPDATE first so that an exclusive lock is placed on this row of COMPANY_ID and no one else can update or read this row until pbUpdate commits the transaction.

Had I executed 'SELECT ID INTO :dfId FROM COMPANY_ID' before the UPDATE operation, using RL isolation level causes all shared locks to be

released by the time control is returned to the client machine. Another user might execute the SELECT statement before this user executes the UPDATE statement. As a result, both the users will assign the same ID to their new records. The ID in COMPANY_ID would be correctly incremented by 2 because of two UPDATE statements executed by the two users.

SqlImmediate

As you might have noticed, I use SqlImmediate for these two operations. SqlImmediate prepares and executes a SQL statement. SqlImmediate actually performs a SqlConnect for a hidden sql handle, a SqlPrepare, a SqlExecute, and for SELECT statements, a SqlFetchNext. The first time you call SqlImmediate, SQLWindows performs all of these functions. On later calls, only some of these functions are performed. For example, if the (hidden) sql handle is still connected to a database, SQLWindows does not perform a SqlConnect. If the SQL statement to compile is the same statement as that used by the last SqlImmediate call, SQLWindows does not perform a SqlPrepare. You can use SqlImmediate with INSERT, UPDATE, DELETE, and other non-query SQL commands. You can use SqlImmediate with a SELECT statement if you expect that the statement only returns one row. While connecting the hidden sql handle, SQLWindows uses the values of the SqlDatabase, SqlUser, and SqlPassword system variables to connect to a database.

Here is the INSERT statement I use to insert the record in the table:

```
'INSERT INTO COMPANY (ID, NAME, PHONE, FAX, ADDR1, ADDR2,
CITY, STATE, ZIP, COUNTRY) VALUES (:dfId, :dfName, dfPhone,
:dfFax, :dfAddr1, :dfAddr2, :dfCity, :dfState, :dfZIP,
:dfCountry)'
```

If everything goes well, I call SqlCommit to commit the transaction. Although I call SqlCommit for hSqlInsert, it commits *all* of the SQL transaction's cursors that are connected to the same database.

Updating an Existing Record

The operation for an UPDATE is straight forward. Here is the SQL statement for the update operation:

```
'UPDATE COMPANY SET NAME=:dfName, PHONE=:dfPhone,
FAX=:dfFax, ADDR1=:dfAddr1, ADDR2=:dfAddr2, CITY=:dfCity,
```

```
STATE=:dfState, ZIP=:dfZIP,  COUNTRY=:dfCountry  WHERE
ROWID=:dfRowid'
```

Updating a Record in a Multi-User Environment

Once again, I used ROWID so this update operation does not overwrite someone else's update to the same row. Let's see how that could happen if I did not use the ROWID. Again, let's assume there are two users; USER1 and USER2. USER1 wants to change the phone number and fax number of company with ID=10. USER2 wants to change the address of the same company.

USER1	USER2
1. Fetches row with ID=10.	1. Fetches row with ID=10.
2. Changes phone number and fax number and updates. Since USER2 has released the shared lock, USER1 is able to acquire exclusive lock and also commit the transaction.	
	2. Changes the address and updates. By now, USER1 has committed so there are no locks on the record. USER2 is able to acquire exclusive lock and update the record. USER2 just overwrote the changes made by USER1! As a result, row with ID=10 contains new address but old phone and fax numbers.

Let me show you how using ROWID can save USER2 from making this mistake.

USER1	USER2
1. Fetches row with ID=10. The ROWID, say, ROWID1 is also fetched.	1. Fetches row with ID=10. The ROWID (ROWID1) is also fetched.

USER1	USER2
2. Changes phone and fax numbers and updates row with ROWID = ROWID1. Since USER2 has released the shared lock, USER1 is able to acquire exclusive lock and also commit the transaction. But since the row has been modified, SQLBase changes the ROWID to, say, ROWID2.	
	2. Changes the address for ID=10 and ROWID = ROWID1 and tries to update. It issues an UPDATE statement to update the row with ROWID = ROWID1. But there is no such row in the table and the update operation fails!

Undoing (Discarding) Changes

If a user modifies an existing record or inserts information about a new record after pressing the Insert push button, any changes made can be undone as long as the user has not pressed the Update push button to commit the transaction. Simply fetching the row again reverses any changes. To remember the row number of the current row, I maintain a variable nRowNumber. pbFirst sets it to 0, pbLast to the number of rows in the result set, and pbPrev and pbNext decrement it and increment it respectively.

```
Pushbutton: pbUndo
 Title: Undo
 Message Actions
  On SAM_Click
   If not bFormDirty and not SalIsNull( dfId )
    ! Not an insert and no changes to fields
    Call SalMessageBeep( 0 )
    Call SalMessageBox(
      'No changes to this record. Nothing to undo.',
      'Error', MB_Ok | MB_IconExclamation )
```

```
      Else
       ! Mark the form as not dirty (no changes yet)
       Set bFormDirty = FALSE
       ! Now re-fetch the same row of the result set.
       Call SqlFetchRow( hSqlSelect, nRowNumber, nFetchResult )
       Call MessageBoxIfFetchError( nFetchResult )
     On SAM_Create
      ! Set the picture for the push button using the resource
      Call SalPicSet( hWndItem, resUndo, PIC_FormatBitmap )
```

Listing 3.13 The pbUndo push button.

Exiting from the Application

When a user wants to exit the application, he or she can choose Close from the system menu, Exit from the File menu or simply press the Exit push button on the toolbar. They all send or post a SAM_Close message to the form window. As seen in Listing 3.4, the form window makes the decision based on whether the user wants to lose any changes if there are any.

```
Pushbutton: pbExit
 Title: Exit
 Message Actions
  On SAM_Click
   Call SalPostMsg( hWndForm, SAM_Close, 0, 0 )
  On SAM_Create
   ! Set the picture for the push button using the resource
   Call SalPicSet( hWndItem, resExit, PIC_FormatBitmap )
```

Listing 3.14 The pbExit push button.

Named Transactions

Normally, the scope of a transaction is *all connections* that an application makes to a given database and user name. A transaction processing operation (such as COMMIT, ROLLBACK, isolation level change, etc.) performed by one transaction does not affect operations performed by other transactions.

Beginning with SQLWindows 5, named transactions let you modularize transactions. For example, you can have:

- Multiple connections to the same database/user name pair that are in different transactions.

- Multiple applications in the same transaction. Named transactions used in this way are called *shared Sql Handles*.

  When you use named transactions with multiple applications, you save:

  - The time to connect new Sql Handles.
  - The memory overhead of maintaining many open Sql Handles.

You cannot use named transactions when you connect through ODBC.

Let me show you the functions to use with named transactions along with their brief description.

SqlConnectTransaction

```
bOk = SqlConnectTransaction( hSql, strTransactionName )
```

This function is like SqlConnect, except that you specify an additional parameter, strTransactionName. The name you specify in strTransactionName associates the connection with the transaction. You can give transactions any names that you want. Transaction names are case sensitive.

When you connect Sql Handles with different transaction names, operations (such as COMMIT or ROLLBACK) you perform for one transaction name do not affect operations that you perform for another transaction name.

When you call SqlConnect, SQLWindows uses an implied transaction name that is hidden.

SqlSharedSet

```
bOk = SqlSharedSet( hSql )
```

This function makes a Sql Handle sharable. SqlSharedSet tells SQLWindows that other applications can use this Sql Handle. You must have connected the Sql Handle with SqlConnectTransaction. SqlSharedSet returns FALSE if the Sql Handle is invalid or does not belong to a named transaction.

SqlSharedAcquire

```
bOk = SqlSharedAcquire( hSql, strTransactionName )
```

Call this function to acquire and use a shared Sql Handle. The Sql Handle must have been connected with SqlConnectTransaction and made sharable with

SqlSharedSet. The parameter strTransactionName must match what the owner application specified when it called SqlConnectTransaction (the string is case sensitive).

SqlSharedRelease

This function releases a shared Sql Handle.

```
bOk = SqlSharedRelease( hSql )
```

This function releases a shared Sql Handle. Call this function in a receiver application when you are finished using the Sql Handle. This function does not disconnect the Sql Handle. SqlSharedRelease returns FALSE if the Sql Handle is invalid, does not belong to a named transaction, or was not made sharable with SqlSharedSet.

4

Object-Oriented Programming

Do you sometimes wonder what object-oriented programming (OOP) is all about? Do you often ask yourself where in your business applications you can make use of OOP? In this chapter, I will answer both these questions. I will choose some needs of a typical business application development environment and illustrate how these needs could be met by using OOP. One reason I am discussing OOP so early in the book is that I will use the concepts in later chapters. Once you learn the concepts, you will begin thinking in terms of object-orientation and often wonder how you have done programming without OOP so far!

With SQLWindows, you don't have to choose between all or nothing. You can start using OOP concepts in your existing applications. In new applications, you can start implementing some OOP concepts initially and add more as time permits and as you feel more comfortable.

Software 'Manufacture'

We all understand how a car is manufactured—there is a team of engineers with a vision of what kind of car they want to build. This team carefully designs all the parts and how they fit together. Several sub-teams then work on each of these parts. These teams have complete freedom to use any materials and how to actually manufacture these parts as long as they meet the specifications set by the designers of the car. Finally, all these various parts are assembled together to make the car. Since all these parts had been designed to work with each other, no problems occur at the assembly line.

Could writing a software application be a similar process? Until very recently, software development has been more of an art than science – mostly because of the lack of tools and the lack of a mechanism that facilitates programming and assembling various parts. Object-oriented programming is a way of writing

software which takes us in that direction. Let us see how OOP takes us in the direction of parts and assembly.

Class – Base Component of OOP

The "parts and assembly" way to manufacture software entails using *classes*, the base components of object-oriented programming. A class is a piece of program which hides the implementation details from the user of the class (an application developer). The class provides a set of methods (such as functions) which the application developer can invoke at the time of assembly (writing an application). The application developer does not have to know how the class has been implemented by the creator of the class. He or she only needs to know the methods that provide an interface to this class. The application programmer can potentially use several classes created by several different programmers. Some of the classes may be created by in-house developers whereas others may be purchased from third party vendors.

Designing Frequently Used Classes

One way to become an efficient programmer is to identify the pieces of programs that you need often and create classes for them. You can then use these classes during application development. I use OOP to design some handy classes.

Auto Entry Data Field Class—clsDfAutoEntry

As an example, let us define a data field class clsDfAutoEntry which monitors every stroke of the key into that field and as soon as the maximum number of characters (as defined using the customizer) have been typed into it, it moves the focus to the next object in the defined tab order. For applications such as online order entry, this greatly increases the efficiency of the sales person entering a new order while talking with the customer on the phone. The sales person does not have to press the TAB key after each field. This is useful when the fields are of fixed size such as part numbers. While using this class, it is not important to know how this class has been implemented but let's look at some of the relevant portions of the code:

```
Application Description: AUTOENTR.APL
  Chapter 4
  Object-Oriented Programming
  Power Programming with SQLWindows
  by Rajesh Lalwani.
```

```
Copyright (c) 1994 by Gupta Corporation.
All rights reserved.
Constants
 User
  ! Define some Microsoft Windows constants
  Number: WM_USER = 0x0400
  Number: EM_SETSEL = WM_USER+1
  Number: WM_NEXTDLGCTL = 0x0028
Class Definitions
 Data Field Class: clsDfAutoEntry
  List in Tool Palette? Yes
  Description: This data field class will automatically
    set focus to the next control in the
    TAB order when it reaches its maximum
    length as set in the customizer.
  Instance Variables
   Number: nMaxLength
  Message Actions
   On SAM_Create
    ! Let us find out the length set in the customizer
      and remember it in the instance variable nMaxLength.
    Set nMaxLength = SalGetMaxDataLength( hWndItem )
   On SAM_AnyEdit
    If SalStrLength( MyValue ) = nMaxLength
     ! Send WM_NEXTDLGCTL to the form so that focus
       will shift to the next control in the TAB order.
     Call SalSendMsg( hWndForm, WM_NEXTDLGCTL, 0, 0 )
   On SAM_SetFocus
    ! Select the entire contents of the data field by
      sending EM_SETSEL message to the data field. lParam
      has begin=0x0000 and end=0xFFFF.
    Call SalSendMsg( hWndItem, EM_SETSEL, 0, 0xFFFF0000 )
```

Listing 4.1 Auto Entry Data Field Class – clsDfAutoEntry.

Think of the above *class* definition as the blue print of a part. There are specifications on the paper but no actual, physical part yet. An actual, physical part which is constructed from the blue print is called an *object* or an *instance* of the object class.

SalGetMaxDataLength

SalGetMaxDataLength returns the maximum length of a data field, multiline text field, or table window column. You can use this function before assigning a

value to any of these objects to ensure that the value fits. The return value is a number that specifies the maximum length of the object. A length of DW_Default (-1) indicates that the object was declared with a length of Default. In this case, I call this function once, at the time the object is created, to get the maximum length as defined in the customizer. I store this away in an instance variable called nMaxLength so that there is no need to call SalGetMaxDataLength again.

Instance Variable—nMaxLength

Instance variables in a class are replicated for each object that you create. Each object has its own private copy of an instance variable. While designing a form window, if you place two data fields derived from clsDfAutoEntry, both these data fields will have their own copy of nMaxLength. For one data field, its value may be 4 while for the other data field, it may be 5.

SAM_AnyEdit

SAM_AnyEdit is sent to a combo box, data field, multiline text field, or table window column whenever you make a change to that object's value. The object receives a SAM_AnyEdit message on every key stroke. By processing SAM_AnyEdit messages, an application can check an object's value as it changes. I use this message to find out the number of characters entered so far by calling SalStrLength.

SalStrLength and SalStrGetBufferLength

SalStrLength returns a string's length. Strings are stored internally in SQLWindows with a null termination character. The null terminator is not included in the length. This is slightly different from the behavior of SalStrGetBufferLength. This function returns the current buffer length of a string. SQLWindows stores string variables in buffers. The buffer length is used when handling binary data and when calling functions defined in an external dynamic link library (DLL). The buffer length is always larger than or equal to the the string length.

MyValue

Note the use of MyValue in the above class definition. You can use this variable in place of an object name in SAL statements within a class. It lets you set or retrieve the window value of the current object from within a window class where no object name is available. The serial number on a part corresponds to

the window value of the push button in the above class definition. Since the window value (serial number) of the instances (parts) derived (constructed) from the clsDfAutoEntry class (blue print) is not available at designtime, MyValue serves the purpose.

SAM_SetFocus

SAM_SetFocus is sent to a check box, combo box, data field, list box, multiline text field, push button, radio button, scroll bar, or table window column when it receives the input focus. You can process the SAM_SetFocus message and initialize actions to take place when the user enters an object. You should avoid calling functions that can change the focus (such as SalMessageBox, and SalModalDialog) while processing a SAM_SetFocus message. In this case, when the data field receives SAM_SetFocus, I send EM_SETSEL to the data field itself so that the entire contents of the data field are selected and highlighted.

Listing a Data Field Class in the Tool Palette

By default the class clsDfAutoEntry would be listed in the tool palette when you have selected the data field button. See Figure 4.1. If you want to list a class derived from certain classes but not the intermediate classes, specify 'List in Tool Palette' to be No in the Customizer for the intermediate classes.

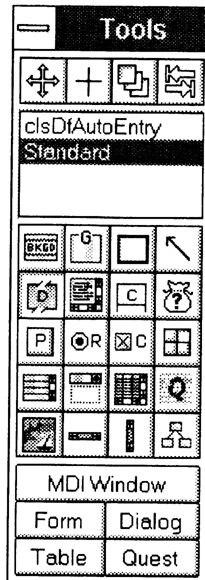

Figure 4.1 User-defined data field class is listed in the tool palette.

Data Field Class of Data Type Number—clsDfNumber

Let's create another handy class. The standard data field in SQLWindows is defined to be of the data type String. If you are like me, even when you want the data field to be of the type Number, you often forget to change the data type; resulting in some funny behavior later. To avoid this, you can define a data field class called clsDfNumber. The data field class clsDfNumber defines the data type to be Number.

Defining New Classes from Previously Defined Classes

This is one area where OOP can be better than the blue print and parts situation. Let's say, we have the blue print of a steering wheel but now we want a steering wheel with an airbag. What are the options? One, we can create a new blue print from scratch. Two, we can take the blue print of the steering wheel and modify it to contain an air bag too. The second option is clearly better than the first one because we can make use of the work done earlier. But is it the best? Probably not; because, if later the engineers working on the design of the steering

wheel (without an air bag) come up with a better design, these changes are not automatically reflected in the design of the steering wheel with an air bag. Fortunately, in software development, you can do better than that using the *inheritance* feature of OOP.

You can define a new class of data field, for example, clsDfCurrency from the class clsDfNumber. clsDfCurrency is derived from clsDfNumber so it *inherits* all of the properties including the data type (Number). In addition, it defines the format of the data field to be of the type Currency. Now, if you change some property of clsDfNumber, for example, background color, it is automatically reflected in the clsDfCurrency class also!

In the previous example, since clsDfCurrency is derived from clsDfNumber, clsDfNumber is called the *base* class or *parent* class of clsDfCurrency.

If you derive another class, for example, clsDf2, from clsDfAutoEntry and this class also defines an instance variable with the same name nMaxLength, the instance variable nMaxLength of clsDf2 hides the same of clsDfAutoEntry. nMaxLength of clsAutoEntry is only hidden, you can still access it by qualifying the name of the instance variable with the name of the base class – clsDfAutoEntry.nMaxLength. Using nMaxLength alone will refer to the instance variable of clsDf2. To remove any ambiguity, you can always use clsDf2.nMaxLength.

Deriving New Classes from Multiple Base Classes

Let's define a new data field class called clsDfAutoEntryNumber. This class has been derived from both clsDfAutoEntry and clsDfNumber defined earlier. This class has the properties of both the classes from which it was derived: the data type of the data field is Number instead of the default String and it automatically sets focus to the next field defined in the TAB order when it reaches its maximum length as set in the customizer. This is an example of *multiple inheritance.*

In case of multiple inheritance, if more than one base class defines or inherits the same instance variable and *if you refer to the instance variable,* you get an error during compilation. You can eliminate the ambiguity by qualifying the name of the instance variable with the name of a class.

Parts and Assembly Case Study—Browse Screen

I am sure every application developer has created screens to browse through records. These screens display the first record and have push buttons which users can use to go to the last, next, previous or first record. In this example, I have initially identified two 'parts' that we can pre-fabricate and use again and again whenever such a browse screen is to be created. In SQLWindows, classes can be either visual classes such as a push button class, a form window class, etc. or non-visual classes called *functional classes*. For this particular case, I would create one of each type: a functional class called clsSqlHandleSelect and a form window class called clsFrmBrowse.

Designing Parts

The first class is a functional class called clsSqlHandleSelect which essentially defines a sql handle (cursor) for connection to the database. When I began designing this class, I realized it may be more useful for the future to split the functionality in two classes instead: clsSqlHandle and clsSqlHandleSelect which is derived from clsSqlHandle. The basic idea is that clsSqlHandle will provide the functionality needed by all SQL statements not just SELECT. clsSqlHandleSelect will then inherit all this functionality and add new features to address the needs of a SELECT statement such as going to the first, next record etc. It is important, however, not to get carried away with the idea of designing generalized classes. You may end up with too many classes that are never used in the future. Also, you have to strike a balance between looking at future versus finishing the current project on time.

Sql Handle Class—clsSqlHandle

clsSqlHandle defines following instance variables: Sql Handle: sqlHandle, String: strSQLStatement, Boolean: bConnected. clsSqlHandle provides the following member functions:

InitializeClass: It initializes the *class variables* strSqlDatabase, strSqlUser, and strSqlPassword. It is assumed that the application connects to one database only. SQLWindows has a concept of class variables. These variables are associated with the class which defines them and are accessible to all the classes that are derived from this class or instances of such classes. They are different from instance variables in that they are shared by all instances – when one instance

changes a class variable, other instances see this new value. In case of instance variables, each instance gets its own copy of the instance variable. Since it was assumed that the application would only connect to one database, it made sense to define the database name, user name and password as class variables so if clsSqlHandle is used more than once, it's not necessary to initialize these variables again.

```
Application Description: SQLHANDL.APL
  Chapter 4
  Object-Oriented Programming
  Power Programming with SQLWindows
  by Rajesh Lalwani.
  Copyright (c) 1994 by Gupta Corporation.
  All rights reserved.
  Class definitions for clsSqlHandle and clsSqlHandleSelect.
  Class Definitions
   Functional Class: clsSqlHandle
    Description: This is the base class for all Sql Handle
      classes. Its InitializeClass() member function should be
      called to initialize the database name, user name, and
      password.  It is assumed that the application connects to
      one database only.
    Class Variables
     ! Class variables for database, user name, and password.
     String: strSqlDatabase
     String: strSqlUser
     String: strSqlPassword
    Instance Variables
     ! variables that each instance derived from this class would
       get.
     Sql Handle: sqlHandle
     String: strSQLStatement
     Boolean: bConnected
    Functions
     Function: InitializeClass
      Description: This function should be called once
        per application. It initializes the class variables
        strSqlDatabase,strSqlUser, and strSqlPassword.
      Parameters
       String: strSqlDatabaseParm
       String: strSqlUserParm
       String: strSqlPasswordParm
      Actions
       ! Set the values of the class variables.
```

```
    Set strSqlDatabase = strSqlDatabaseParm
    Set strSqlUser = strSqlUserParm
    Set strSqlPassword = strSqlPasswordParm
Function: Initialize
  Description: This function should be called once per
    instance to initialize the SQL statement.  It is assumed
    that the InitializeClass function has been called before.
  Parameters
   String: strSQLStatementParm
  Actions
   ! Set the value of the instance variable for remembering
   ! the SQL statement.
   Set strSQLStatement = strSQLStatementParm
Function: Error
  Description: This function displays the error text
    in a message box. A class derived from clsSqlHandle
    can override this function by defining its own Error
    function.
  Parameters
   String: strMessage
  Actions
   Call SalMessageBox( strMessage, 'clsSqlHandle.Error',
     MB_Ok | MB_IconStop )
Function: Connect
  Description: This function calls SqlConnect.
    It uses the class variables to set the system variables
    SqlDatabase, SqlUser and SqlPassword. It returns TRUE
    if SqlConnect was successful, FALSE otherwise.
  Returns
   Boolean:
  Local variables
   String: strSqlDatabaseTemp
   String: strSqlUserTemp
   String: strSqlPasswordTemp
  Actions
   ! Remember the old values of the system variables
   Set strSqlDatabaseTemp = SqlDatabase
   Set strSqlUserTemp = SqlUser
   Set strSqlPasswordTemp = SqlPassword
   ! Set the values of the system variables using
   ! the class variables.
   Set SqlDatabase = strSqlDatabase
   Set SqlUser = strSqlUser
   Set SqlPassword = strSqlPassword
   ! Define a local error handler in case of a SQL error.
```

```
      When SqlError
        ! Call the Error function in a late-bound fashion so
          that if another Error function has been defined down
          the inheritance chain, that function would be called
          instead.
        Call ..Error('Could not SqlConnect on Database : '
          || strSqlDatabase || ' for User : ' || strSqlUser)
        ! Return FALSE so that the SQL function would return
          FALSE to its caller.
        Return FALSE
      ! Call SqlConnect
      Set bConnected = SqlConnect( sqlHandle )
      ! Reset the values of the system variables using the temp
        variables.
      Set SqlDatabase = strSqlDatabaseTemp
      Set SqlUser = strSqlUserTemp
      Set SqlPassword = strSqlPasswordTemp
      ! Return the success/failure of the operation
      Return bConnected
  Function: SetIsolationLevel
    Description: This function can be used to set
      isolation level for all cursors of the application.
      Read Repeatability is the default setting for SQLWindows.
    Returns
      Boolean:
    Parameters
      String: strIsolation
    Actions
      Return SqlSetIsolationLevel( sqlHandle, strIsolation )
  Function: Prepare
    Description: This function prepares the SQL statement.
    Returns
      Boolean:
    Local variables
      Boolean: bPrepared
    Actions
      If bConnected
        When SqlError
          ! Call the Error function in a late-bound fashion so
            that if another Error function has been defined down
            the inheritance chain, that function would be called
            instead.
          Call ..Error ('Could not prepare: ' || strSQLStatement)
          ! Return FALSE so that the SQL function would return
            FALSE to its caller.
```

```
      Return FALSE
     Set bPrepared = SqlPrepare( sqlHandle, strSQLStatement )
    Else
     Set bPrepared = FALSE
    Return bPrepared
Function: Execute
  Description: This function is called to execute the SQL
    statement which has been prepared earlier by calling
    Prepare function.
  Returns
   Boolean:
  Local variables
   Boolean: bExecuted
  Actions
   When SqlError
     ! Call the Error function in a late-bound fashion so
       that if another Error function has been defined down
       the inheritance chain, that function would be called
       instead.
     Call ..Error('Could not execute: ' || strSQLStatement)
     ! Return FALSE so that the SQL function would return
       FALSE to its caller.
     Return FALSE
   Set bExecuted = SqlExecute( sqlHandle )
   Return bExecuted
Function: Commit
  Description: This function is called to commit the
    transaction.
  Returns
   Boolean:
  Local variables
   Boolean: bCommited
  Actions
   When SqlError
     ! Call the Error function in a late-bound fashion so
       that if another Error function has been defined down
       the inheritance chain, that function would be called
       instead.
     Call ..Error('Could not commit: ' || strSQLStatement)
     ! Return FALSE so that the SQL function would return
       FALSE to its caller.
     Return FALSE
   Set bCommited = SqlCommit( sqlHandle )
   Return bCommited
```

```
Function: Disconnect
 Description: If the Sql Handle is connected, it is
   disconnected it by calling SqlDisconnect.
 Returns
  Boolean:
 Actions
  If bConnected
   When SqlError
     ! Call the Error function in a late-bound fashion so
       that if another Error function has been defined down
       the inheritance chain, that function would be called
       instead.
     Call ..Error
        ( 'Could not disconnect: ' || strSqlDatabase )
     ! Return FALSE so that the SQL function would return
       FALSE to its caller.
     Return FALSE
   Set bConnected = not SqlDisconnect( sqlHandle )
   Return not bConnected
  Else
   ! The Sql Handle was never connected.
   Return TRUE
```

Listing 4.2 Sql Handle class clsSqlHandle.

Initialize: This function should be called once per instance to initialize the SQL statement. In this example, I call this function with the exact SELECT statement when I design a browse screen from clsFrmBrowse.

Error: This function displays the error text in a message box. A class or an instance derived from clsSqlHandle can override this function by defining its own Error function. If another Error function is defined by the class or instance derived from clsSqlHandle, that function is called instead because within clsSqlHandle, Error is called in a *late bound* fashion indicated by two periods before the function name: Call ..Error(). In case of a late bound call, a check is made at runtime to see which Error function is to be used. This provides some flexibility to the users of the class. On the other hand, late bound calls are more expensive. That is why, SAL has chosen early binding as the default. Late binding is also known as dynamic binding. As you will see shortly, clsFrmBrowse, indeed, defines its own Error function which overrides the Error function defined by clsSqlHandle. This is indicated by strikethrough style in Figure 4.3.

Connect: This function calls SqlConnect. It uses the class variables to set the system variables SqlDatabase, SqlUser, and SqlPassword. Note the use of a local error handler – When SqlError statement.

Local Error Processing—When SqlError Statement

When you call a SQL function and it fails (returns FALSE), an error message is sent to the application. Normally, this initiates default error processing. SQLWindows lets you control the way errors are handled. You can add a When SqlError statement to any code section (for example, an object's Message Actions section). The way in which an error is handled is part of the error processing hierarchy. SQLWindows starts from the inside of your program and works outward; that is, it starts the error process from a local section (where the error occurred) and works outward to the global sections. A When SqlError statement must precede any SQL statements that you want handled locally in the case of an error. Also, the When SqlError statement must be at the same level in the application outline as the SQL statement which produced the error.

SetIsolationLevel. This function is called to set isolation level for all cursors of the application. Read Repeatability is the default setting for SQLBase.

Prepare: This function prepares the SQL statement.

Execute: This function is called to execute the SQL statement which has been prepared earlier by calling Prepare function. Later, you will see that clsSqlHandleSelect overrides this function by defining its own Execute function. Again, this is indicated by strikethrough style in Figure 4.3. Actually, clsSqlHandleSelect extends this function by calling Execute of clsSqlHandle first.

Commit: This function calls SqlCommit to commit the transaction.

Disconnect: If the Sql Handle is connected, it is disconnected by calling SqlDisconnect.

Sql Handle Class for SELECT statements—clsSqlHandleSelect

clsSqlHandleSelect defines one additional instance variable Number: nResultSetCount to keep track of the total number of rows in the result set. This is used by a member function – Last. clsSqlHandleSelect provides the following member functions:

Execute: This function is called to execute the SELECT statement which has been prepared earlier by calling Prepare function. This function overrides (actually extends) the Execute function defined in the base class clsSqlHandle. It first calls the Execute function of the clsSqlHandle class to execute the SELECT statement by qualifying the function name with the class name – clsSqlHandle.Execute. It then calls SqlGetResultSetCount to set the value of nResultSetCount – instance variable of clsSqlHandleSelect.

SqlGetResultSetCount counts the rows in a result set by building the result set. Servers such as SQLBase provide the number of rows in the result set without actually fetching each row. But for some databases, calling SqlGetResultSetCount can be a time-consuming operation. You can choose to initialize the instance variable nResultSetCount only when the member function Last is called the first time.

MessageBoxIfFetchError. This function displays a message box in case First, Next, Previous, Last, FetchRow functions return FETCH_EOF, FETCH_Delete, or FETCH_Update. FETCH_EOF is sent in two cases: 1) when the user is at the last record and tries to go to a next record; 2) when the user is at the first record and attempts to go to a previous record.

SQLWindows does not distinguish among its member functions like the way C++ does; C++ has public, private, and protected member functions in a class. If SQLWindows did distinguish, I would have declared MessageBoxIfFetchError as a private function to be available only within this class definition. I use this function only locally to process the fetch return code and display an appropriate message.

```
Functional Class: clsSqlHandleSelect
 Description: This class is derived from clsSqlHandle.
   This class can be used to define a SqlHandle which is
   going to be used to do a SELECT and browse (First record,
   Last record, Next record, Previous record,specific record)
   through the result set. The instance can define the specific
   SELECT statement by calling the Initialize() function.
 Derived From
  Class: clsSqlHandle
 Instance Variables
  ! It has an additional instance variable to remember the
    number of rows in the result set created by the SELECT.
  Number: nResultSetCount
```

```
Functions
 Function: Execute
  Description: This function is called to execute the
    SELECT statement which has been prepared earlier by
    calling Prepare function. This function overrides
    (actually extends) the Execute function defined in the
    base class clsSqlHandle.
  Returns
   Boolean:
  Local variables
   Boolean: bSuccess
  Actions
   ! First call the Execute function of the base class
     clsSqlHandle.
   Set bSuccess = clsSqlHandle.Execute( )
   If bSuccess
    When SqlError
     ! Call the Error function in a late-bound fashion so
       that if another Error function has been defined down
       the inheritance chain, that function would be called
       instead.
     Call ..Error('Could not get result set count for: '
       || strSQLStatement)
     Set nResultSetCount = 0
     ! Return FALSE so that the SQL function would return
       FALSE to its caller.
     Return FALSE
    Set bSuccess = SqlGetResultSetCount( sqlHandle,
      nResultSetCount )
   Return bSuccess
 Function: MessageBoxIfFetchError
  Description: This function displays a message box
    in case First, Next, Previous, Last return FETCH_EOF,
    FETCH_Delete, or FETCH_Update.
  Returns
   Boolean:
  Parameters
   Number: nInd
  Local variables
   String: strMessage
  Actions
   Select Case nInd
    Case FETCH_EOF
     Set strMessage = 'Reached the end.'
     Break
```

```
    Case FETCH_Update
     Set strMessage = 'The record has been updated.'
     Break
    Case FETCH_Delete
     Set strMessage = 'The record has been deleted.'
     Break
    Default
     Return TRUE
   Call SalMessageBeep( 0 )
   Call SalMessageBox(strMessage, 'FETCH Information', MB_Ok )
   Return FALSE
Function: First
 Description: This function is used to go to the
   first row in the result set.
 Returns
  Boolean:
 Parameters
  Receive Number: nInd
 Actions
  Return FetchRow( 0, nInd)
Function: Next
 Description: This function is used to fetch the
   next row in the result set.  This function or First
   must be called immediately after Execute to position to
   the first row.
 Returns
  Boolean:
 Parameters
  Receive Number: nInd
 Local variables
  Boolean: bSuccess
 Actions
  Set bSuccess = FALSE
  When SqlError
   ! Call the Error function in a late-bound fashion.
   Call ..Error('Cannot Fetch Next Record')
   ! Return FALSE so that the SQL function would return
     FALSE to its caller.
   Return FALSE
  Set bSuccess = SqlFetchNext( sqlHandle, nInd )
  Call MessageBoxIfFetchError( nInd )
  Return bSuccess
Function: Previous
 Description: This function is used to fetch the
   previous row in the result set.
```

```
      Returns
       Boolean:
      Parameters
       Receive Number: nInd
      Local variables
       Boolean: bSuccess
      Actions
       Set bSuccess = FALSE
       When SqlError
        ! Call the Error function in a late-bound fashion.
        Call ..Error('Cannot Fetch Previous Record')
        ! Return FALSE so that the SQL function would return
          FALSE to its caller.
        Return FALSE
       Set bSuccess = SqlFetchPrevious( sqlHandle, nInd )
       Call MessageBoxIfFetchError( nInd )
       Return bSuccess
     Function: Last
      Description: This function is used to go to the
        last row in the result set.
      Returns
       Boolean:
      Parameters
       Receive Number: nInd
      Actions
       ! 0-based row number
       Return FetchRow( nResultSetCount-1, nInd)
     Function: FetchRow
      Description: This function is used to fetch a
        specific row in the result set.
      Returns
       Boolean:
      Parameters
       Number: nRowParm
       Receive Number: nInd
      Local variables
       Boolean: bSuccess
      Actions
       Set bSuccess = FALSE
       When SqlError
        ! Call the Error function in a late-bound fashion.
        Call ..Error('Cannot Fetch Record# ' ||
          SalNumberToStrX( nRowParm, 0 ))
        ! Return FALSE so that the SQL function would return
          FALSE to its caller.
```

```
         Return FALSE
        Set bSuccess = SqlFetchRow( sqlHandle, nRowParm, nInd )
        Call MessageBoxIfFetchError( nInd )
        Return bSuccess
```

Listing 4.3 Sql Handle Class for SELECT statements – clsSqlHandleSelect.

First, Previous, Next, Last: These functions are used to go to the first, previous, next, and last rows in the result set. Last makes use of the instance variable nResultSetCount.

FetchRow: This function is used to fetch a specific row in the result set. This function is used by member functions First and Last. SalNumberToStrX converts a number to a string. The second parameter specifies the number of decimal places you want.

Form Window Class for Browse Screens—clsFrmBrowse

clsFrmBrowse is a Form Window class. It is different from the previous two classes in the sense that it is a visual class. Even though it is a visual class, SQLWindows lets you derive it from a non-visual (functional) class. clsFrmBrowse is derived from clsSqlHandleSelect so it inherits all the methods and instance variables of clsSqlHandleSelect. Figure 4.2 shows the template for clsFrmBrowse as seen through SQLWindows Class Editor.

In this case, I have chosen to derive clsFrmBrowse from clsSqlHandleSelect. Alternately, I could define an instance variable, for example, sqlHandleSelect derived from clsSqlHandleSelect. This is the only option if I need more than one instance of clsSqlHandleSelect.

I have chosen to directly inherit from clsSqlHandleSelect because all the variables and member functions of clsSqlHandleSelect become directly available to clsFrmBrowse and any class or screen derived from it. Figure 4.3 shows the three classes designed thus far.

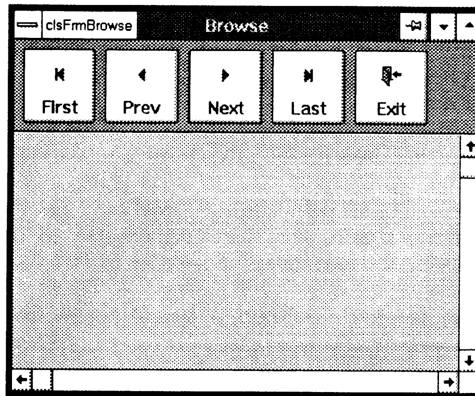

Figure 4.2 clsFrmBrowse as seen through SQLWindows Class Editor.

This Form Window class provides a template for creating browse screens with push buttons for going to the first, previous, next, and last records. When you design a screen based on this class, call Initialize on SAM_Create to initialize the SQL SELECT statement. Note that clsFrmBrowse does not define its own Initialize function – it inherits it from clsSqlHandleSelect which in turn had inherited it from clsSqlHandle.

clsFrmBrowse defines some push buttons as the contents of the toolbar and puts necessary code for SAM_Click event for them. For example, for pbFirst, it calls First function which was defined by clsSqlHandleSelect.

Since it is assumed that the screen derived from this class would have called Initialize for SAM_Create, this class uses SAM_CreateComplete to call Connect, Prepare, Execute, and Next member functions to connect to the database, prepare and execute the SELECT statement, and fetch the first record of the result set.

SQLWindows sends SAM_CreateComplete message to windows with contents (top-level windows, child table windows, and child QuestWindows) after creating the window's children and displaying the window and its children.

Finally, the pbExit push button posts a SAM_Close message to the *hWndForm*. Since we do not know the window handle name or the name of the screen which will be derived from this Form Window class, the reserved word hWndForm would provide us the window handle of the screen at runtime.

hWndForm

This variable takes one of two values:

- When actions are executing from an application's Application Actions section, hWndForm is NULL.
- When actions are executing from a window's Message Actions section, hWndForm is the window handle of the current top level window. This top level window can be a form window, table window, or dialog box to which the current message is sent. The only exception is when the window is a child table window column; in this case, hWndForm is the child table window.

SAM_Destroy

In response to SAM_Destroy, clsFrmBrowse calls member function Disconnect to disconnect from the database.

SAM_Destroy is sent to a top-level window (dialog box, form window, or table window) and then to all of its children just before the windows are destroyed. SAM_Destroy messages are sent after SAM_Close has been sent to the top-level window. For example, if a form window has data fields, SQLWindows sends the messages to the objects in this order:

1. SAM_Close to the form window.
2. SAM_Destroy to the form window.
3. SAM_Destroy to each of the form window's child windows.

After all of the SAM_Destroy messages are sent, the top-level and child windows are destroyed. It is also sent to an MDI window.

```
Application Description: FRMBROWSE.APL
  Chapter 4
  Object-Oriented Programming
  Power Programming with SQLWindows
  by Rajesh Lalwani.
  Copyright (c) 1994 by Gupta Corporation.
  All rights reserved.
  This APL defines a Form Window class clsFrmBrowse.
 Libraries
  File Include: SQLHANDL.APL
```

```
Global Declarations
 Class Definitions
  Form Window Class: clsFrmBrowse
   Title: Browse
   Description:
     This class is a Form Window class to provide a template for
     creating browse screens with push buttons for going to
     first, previous, next and last records. Call Initialize()
     on SAM_Create to initialize the SQL SELECT statement.
   Derived From
    Class: clsSqlHandleSelect
    Contents
     Pushbutton: pbFirst
      Title: First
      Picture File Name: FIRST.BMP
      Picture Transparent Color: Gray
      Message Actions
       On SAM_Click
        Call First( nInd )
     Pushbutton: pbPrevious
      Title: Prev
      Picture File Name: PREV.BMP
      Picture Transparent Color: Gray
      Message Actions
       On SAM_Click
        Call Previous( nInd )
     Pushbutton: pbNext
      Title: Next
      Picture File Name: NEXT.BMP
      Picture Transparent Color: Gray
      Message Actions
       On SAM_Click
        Call Next( nInd )
     Pushbutton: pbLast
      Title: Last
      Picture File Name: LAST.BMP
      Picture Transparent Color: Gray
      Message Actions
       On SAM_Click
        Call Last( nInd )
     Pushbutton: pbExit
      Title: Exit
      Picture File Name: EXIT.BMP
      Picture Transparent Color: Gray
```

```
    Message Actions
     On SAM_Click
      Call SalPostMsg( hWndForm, SAM_Close, 0, 0 )
 Instance Variables
  ! To store the return indicator of fetch functions.
    Not intended as an instance variable. Only a work around
    because could not declare local variables for push
    buttons.
  Number: nInd
 Functions
  Function: Error
   Description: This function overrides the function Error of
     clsSqlHandle.  This function gets the error number of
     the last error, error text, cause and remedy.
   Parameters
    String: strMessage
   Local variables
    Number: nError
   Actions
    ! Get the most recent SQL error for the sqlHandle
    Set nError = SqlError( sqlHandle )
    ! Show the modal dialog box to display the error, cause and
      remedy.
    Call SalModalDialog( dlgSqlError, hWndForm , nError,
       strMessage)
 Message Actions
  On SAM_CreateComplete
   ! Display the first record
   If not (Connect(  ) and
     SetIsolationLevel( 'RL' ) and
     Prepare( ) and
     Execute( ) and
     Next( nInd ))
    Call SalPostMsg( hWndForm, SAM_Close, 0, 0 )
  On SAM_Destroy
   Call Disconnect(  )
```

Listing 4.4 Form Window class for browse screens—clsFrmBrowse.

Notice the use of 'Picture Transparent Color' for the push buttons on the toolbar. If you set this attribute using the customizer for the push button, the push button's background color replaces the color you select wherever it appears in an image. This applies to bitmaps only (*.BMP). For example, if the push button has

a white background and you have specified gray as the picture transparent color using the customizer, gray color in the BMP will be replaced by white.

Error: clsFrmBrowse overrides the Error function which it inherited from clsSqlHandleSelect which, in turn, had inherited it from clsSqlHandle. This new Error function provides a lot more information in case of an error—it provides the message generated by the class, error text, reason, and the remedy on a dialog box dlgSqlError discussed earlier in Chapter 3.

Class: clsSqlHandle	Class: clsSqlHandleSelect	Class: clsFrmBrowse
Class Variables: String: strSqlDatabase String: strSqlUser String: strSqlPassword	Class Variables:	Class Variables:
Instance Variables: Boolean: bConnected Sql Handle: sqlHandle String: strSQLStatement	Instance Variables: Number: nResultSetCount	Instance Variables: Number: nInd

Class: clsSqlHandle	Class: clsSqlHandleSelect	Class: clsFrmBrowse
Methods:	Methods:	Methods:
InitializeClass	Execute	Error
Initialize	MessageBoxIfFetchError	
~~Error~~	Next	SAM_CreateComplete
Connect	Previous	SAM_Destroy
SetIsolationLevel	FetchRow	
Prepare	First	
~~Execute~~	Last	
Commit		
Disconnect		

Figure 4.3 clsFrmBrowse is derived from clsSqlHandleSelect which, in turn, is derived from clsSqlHandle. Strikethrough style indicates that the method has been overridden by a derived class.

Assembly Line

Once we have designed these 'parts', designing an actual screen is very easy. When you want to create a new form, the outline options bar now displays clsFrmBrowse in addition to Form Window in the choices. If you choose clsFrmBrowse, you create a new form (screen) which is derived from clsFrmBrowse. Let's call it frmBrowseGuest. frmBrowseGuest would be an instance of the class clsFrmBrowse. You can place data fields and background text on frmBrowseGuest as usual.

```
Application Description: OOP.APP
  Chapter 4
  Object-Oriented Programming
  Power Programming with SQLWindows
  by Rajesh Lalwani.
  Copyright (c) 1994 by Gupta Corporation.
  All rights reserved.
  This application displays a screen to browse all
  the guests in the GUEST table of the GUPTA database.
```

```
Libraries
 File Include: FRMBROWS.APL
 File Include: SQLHANDL.APL
Form Window: frmBrowseGuest
 Class: clsFrmBrowse
 Title: Browse Records of GUEST Table of GUPTA Database
 Description: This form window is an instance of clsFrmBrowse.
 Menu
  Menu Item: About!
   Menu Actions
    Call SalModalDialog( dlgAbout, hWndForm )
 Contents
  Background Text: Name:
  Data Field: dfName
  Background Text: Room Name:
  Data Field: dfRoomName
  Background Text: Trainer:
  Data Field: dfTrainer
  Background Text: In Weight:
  Data Field: dfInWeight
  Background Text: Target Weight:
  Data Field: dfTargetWeight
 Message Actions
  On SAM_Create
   Call frmBrowseGuest.InitializeClass
     ( 'SPA', 'SYSADM', 'SYSADM' )
   Call frmBrowseGuest.Initialize
     ( 'SELECT NAME, ROOM_NAME, TRAINER, IN_WEIGHT, TARGET_WEIGHT
     INTO :dfName, :dfRoomName, :dfTrainer, :dfInWeight,
     :dfTargetWeight FROM GUEST' )
```

Listing 4.5 Deriving a Screen (Form Window) from clsFrmBrowse.

That's all you need to do. You now have a fully functional application which brings up a screen to display records from GUEST table of GUPTA database. It has a toolbar with push buttons for browsing and an Exit button to exit from this screen. Figure 4.4 shows the screen when this application is run. Notice the title of this screen. Compare it with the title as seen in Figure 4.2. I have chosen to override the title defined in the class clsFrmBrowse.

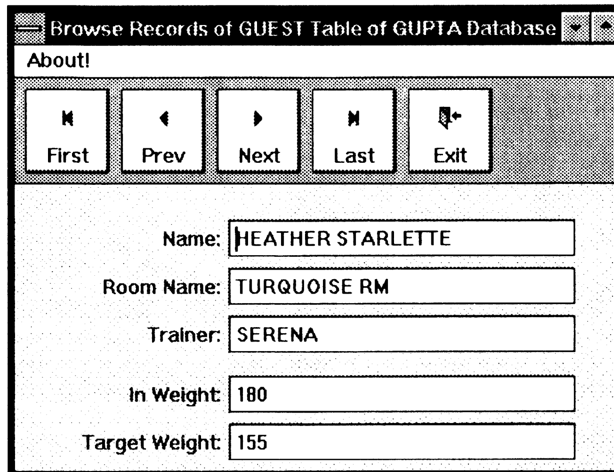

Figure 4.4 An actual screen derived from clsFrmBrowse.

Late Binding versus Early Binding

Let me now show you how using late binding for calling Error function differs from using early binding. Let's assume that I made a mistake while designing frmBrowseGuest—instead of making the datafield dfName of data type String, I made it a datafield of data type Number. When I run this application, the dialog box as shown in Figure 4.5 comes up explaining the message, error text, reason, and the remedy in multi-line text fields. So even though clsSqlHandle defines Error function to display a simple message box, when clsSqlHandleSelect.Next calls ..Error, the Error function defined by clsFrmBrowse is called instead.

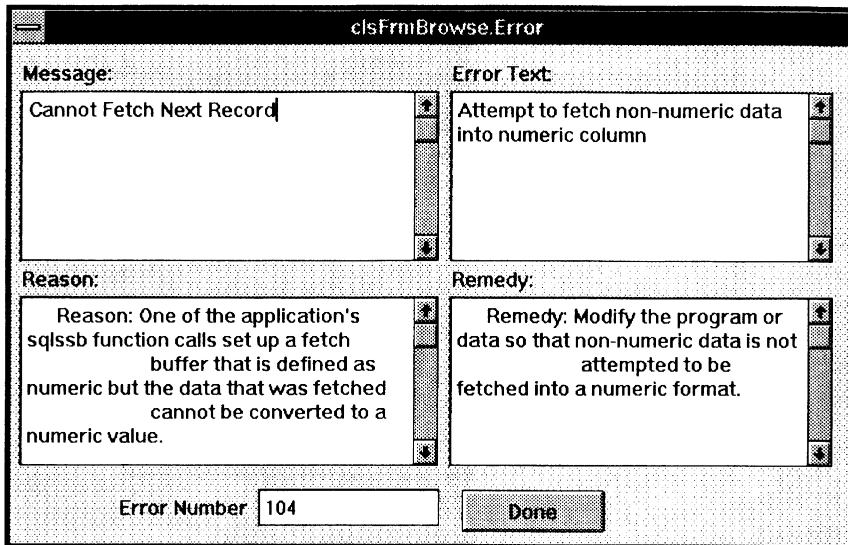

Figure 4.5 Dialog box to indicate the error message, error text, reason and remedy in case dfName was declared of data type Number instead of String.

Had clsFrmBrowse chosen not to write its own Error function, the late bound call ..Error would have essentially been the same as an early bound call and the message box as shown in Figure 4.6 would be displayed. Notice the difference in the titles of the dialog box and the message box.

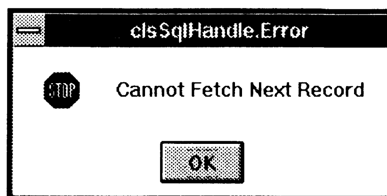

Figure 4.6 Message box to indicate the error message in case dfName was declared of data type Number instead of String.

Corporate Standards for User Interface

One place where corporate MIS departments use OOP is in enforcing corporate standards, particulary in user interface. One such example would be to define a form window class clsWndCorporate which has a standard background color (for example, yellow), corporate logo, and a standard set of push buttons on the toolbar. See Figure 4.7 for clsWndCorporate. All the screens developed by the MIS department are derived from this and hence have the same look and feel.

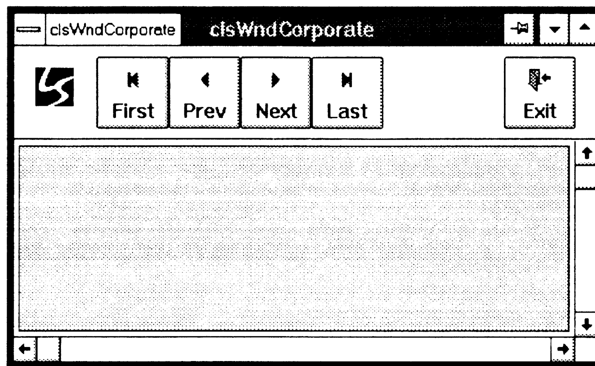

Figure 4.7 clsWndCorporate as a standard for all screens.

Another place where OOP can be used in enforcing corporate standards is to provide a library of data field classes for specific purposes. A non-editable (read-only) data field class may have a yellow background so that whenever an end user sees a yellow datafield on the screen, he or she immediately knows that it is a read-only data field. Similar color coding schemes can be developed for string versus numeric data fields.

Easy Maintenance of Code

It is a very good idea to always have a base class for every visual object used in an application, even if the base class is the same as the default SQLWindows class. Later, it is very easy to ripple the changes through all the objects by making a change in the base class alone. Let's assume that all the form windows were derived from clsFrmCorpBase. When the application was first written, security may not have been an issue. At some point you may want, for security reasons,

to monitor the keyboard activity for each screen and destroy the screen if there has not been any activity for past 20 minutes. Since all the screens were derived from clsFrmCorpBase, it is easy to incorporate this functionality in the class itself. All the screens get this security feature for free!

Hiding Implementation Details

A class is essentially a unit which contains some data (variables) and some methods (functions, event handlers). In a true object oriented environment, the data should be completely hidden from the outside world. The outside world can manipulate this data through the use of the methods provided by the object. The internal details are hidden. This makes it very easy to change the implementation of the methods as long as the interface remains the same.

5

MDI Windows

About MDI Windows

Have you ever written or used applications where you needed to display several screens at a time? Did you wish there was a way to manage them so that you could tile them or minimize them all at once instead of minimizing each screen separately? Well, MDI Windows may be an answer for you. MDI stands for Multiple Document Interface and is a user interface model created by Microsoft to handle multiple documents (screens) in an application. As an example, SQLWindows itself uses it to manage the outline options bar, tool palette, and many views of an outline such as Main, Internal Functions, Classes, External Functions, and Resources. When you choose to minimize the MDI window, everything contained in the MDI window goes with it. MDI windows are used by most Microsoft Windows applications such as Quest, Microsoft Word, and Microsoft Excel.

Managing Phone Numbers and Addresses

In this chapter, I build an application called PHAD.APP (name derived from Phone, Address) to manage:

- Name—salutation such as Mr. or Ms., first name, middle initial, and last name.
- Phone numbers—home phone, work phone, and an alternate phone number which can be used for storing a fax number.
- Company name.
- Address— three lines of street address along with city, state, zip code and country information.
- Important dates—birthday, annivarsary, and one other user specified date which can be used, for example, for the birthday of the spouse.

- Notes.

PHAD.APP uses an MDI window to manage two screens. A form window displays complete information about a person. A user can modify this information, delete this record, or insert a new record. The other screen is a table window which displays all the records in a scrollable table. Each row

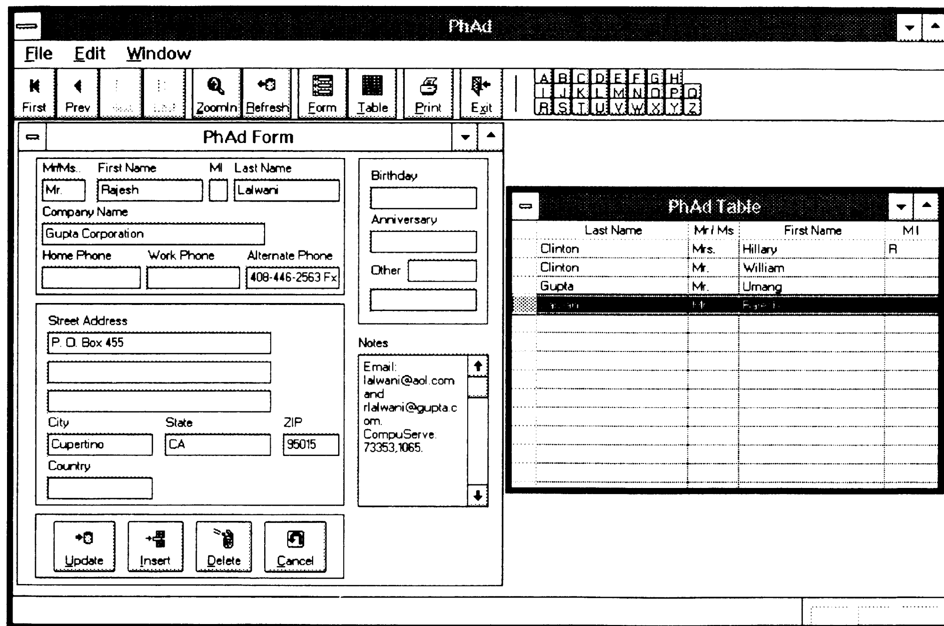

Figure 5.1 PHAD.APP – MDI Window and Two Child Windows.

corresponds to a person. It only displays columns for the last name, salutation such as Mr. or Ms., first name, and middle initial. A user can scroll through the table using the scroll bar or keyboard keys. A user can also select a particular record by clicking on that row. Finally, PHAD.APP provides an index of letters, A through Z, so a user can go directly to a person whose last name starts with that letter. Both the form window and the table window are kept in

synchronization. If the table window has focus on a row for Mr. William Clinton, the form window also displays information about Mr. William Clinton.

You can use PHAD.APP to generate three reports: normal, two columns per page, and mailing labels. The reports can either be printed or viewed on the screen. The topic of generating reports is discussed in detail in Chapter 7.

At this point, I encourage you to import the PHAD table as explained below and run the PHAD.APP application.

Structure of PHAD Table

PHAD.APP uses a table called PHAD to store information about each person. The accompanying disk contains PHAD.SQL which is used to create the table in a database. You can use SQLTalk (or WinTalk) for this. Start SQLTalk and follow these steps:

1. Connect to a database where you want to create the PHAD table.
2. Use the LOAD command to load the PHAD.SQL file. This will create the PHAD table and insert a few records in the table.
3. Disconnect from the database and exit from SQLTalk.

Listing 5.1 shows a typical session.

```
connect phad SYSADM/SYSADM;
load sql a:\phad.sql;
disconnect phad;
exit;
```

Listing 5.1 SQLTalk commands to create a PHAD table.

You should have no problem in loading the PHAD.SQL file. However, if you do, you can create a table called PHAD with the following columns:

```
MRMS CHAR (5)
FIRSTNAME CHAR (20)
MIDDLEINITIAL CHAR (1)
LASTNAME CHAR (20)
COMPANY CHAR (40)
HOMEPHONE CHAR (20)
WORKPHONE CHAR (20)
ALTERNATEPHONE CHAR (20)
```

```
STREET1 CHAR (30)
STREET2 CHAR (30)
STREET3 CHAR (30)
CITY CHAR (20)
STATE CHAR (20)
ZIP CHAR (10)
COUNTRY CHAR (20)
OTHEROCCASION CHAR (10)
BIRTHDAY CHAR (10)
ANNIVERSARY CHAR (10)
OTHERDAY CHAR (10)
NOTES LONG VARCHAR.
```

Architecture of an MDI Window Application

Using an MDI window in an application brings a certain structure to the application code. In comparison with an application that has several top-level windows that communicate with each other, an MDI window application requires some programming discipline in terms of communication among the child windows and the division of responsibilities. In general, there should be no direct communication among the child windows. All the communication should be between the MDI window and a child window.

Architecture of PHAD.APP

PHAD.APP has an MDI window called mdiWindow which contains two other child windows. mdifrmOnePerson is a form window and mditblTable is a table window. I have enabled the accessories of mdiWindow and made the toolbar and status bar visible by using the customizer. Let me show you how different tasks are handled by PHAD.APP internally.

- **Login**

  The SAM_AppStartup message handler of the application brings up the login dialog box dlgLogin and connects one sql handle – hSqlTable for the table window. If the connection is successful, the dialog box sets the system variables SqlDatabase, SqlUser and SqlPassword. These variables are used by the SAM_Create message handler of the form window to initialize the

class variables of user-defined variables hSqlFormSelect, hSqlFormUpdate, hSqlFormDelete, and hSqlFormInsert.

- **Refresh**

  MDI windows posts a PM_Refresh message to itself when the MDI window is created (SAM_Create). When the MDI window receives PM_Refresh, either during creation or later because of the Refresh push button, it simply sends the same message (PM_Refresh) to all its children, the form window and the table window. It is the responsibility of these child windows to take appropriate actions to prepare the necessary SQL statement, and execute it to create the result set.

- **Browsing**

  When a user presses the First, Prev, Next, and Last push buttons on the toolbar of the MDI window, the MDI window simply sends the PM_GoToFirst, PM_GoToPrev, PM_GoToNext, or PM_GoToLast message to all its children. Once again, it is the responsibility of the individual child windows to take appropriate actions.

- **Disabling and Enabling of Push Buttons**

  When a user is looking at the first record, both First and Prev push buttons are disabled. Similarly, when a user is looking at the last record, both Next and Last push buttons are disabled. All these push buttons are enabled at all other times. Ideally, I would prefer the MDI window to maintain the state of these push buttons because they belong to its toolbar. Unfortunately, the MDI window does not know the current location within the result set. In this application, I assign this task to the form window. The form window enables and disables the appropriate push buttons by qualifying push button names with the name of the MDI window.

- **Using Index Push Buttons**

  All index push buttons are instances of a push button class clsIndexPushButton. This class has an instance variable nIndexLetter which

is initialized by each of the 26 instances of the letter they represent. When a push button, say is pressed, it sends a PM_ IndexLetter message to the MDI window and passes the nIndexLetter in wParam. Upon receiving PM_IndexLetter, the MDI window sends PM_IndexLetter to the table window. The table window uses binary search to search for a row with the last name starting with this letter. It returns the row number to the MDI window. Upon receiving this row number, the MDI window posts the PM_GoToLParam message to itself by supplying the row number in the lParam. Upon receiving PM_GoToLParam, the MDI simply sends this message to all its children which in turn fetch that particular record.

- **Clicking on a Row of the Table Window**

  If a user clicks on a row of the table window, either a SAM_Click or SAM_RowHeaderClick message is sent to the table window depending on whether the user clicked on the the row itself or the row header on the left side of the table. The table window gets the row number in the lParam and posts a PM_GoToLParam message to the MDI window which, in turn, sends it to all its the children.

- **Keeping Track of Active Child**

  The ZoomIn push button of the MDI window's toolbar needs to know which of the two child windows is currently active. The MDI window uses a boolean variable bFormActivated to keep track of it. This variable is maintained by the form window. Beginning with Release 4.1, SQLWindows sends a SAM_Activate message to a top-level window when it is activated or deactivated. wParam tells whether it is being activated (wParam = TRUE) or deactivated (wParam = FALSE). The form window processes this message and sets the boolean variable appropriately.

- **Zooming In**

  If the form window is currently active, this push button displays the contents of the control that has the focus on the form window. It is particularly useful for data fields where the data does not fit in its size and for the Notes multiline field. If the table window is currently active, the MDI

window calls a table window function, GetFocusRowStr, to get the salutation, first name, middle initial, and last name concatenated in one string which is then displayed in a dialog box.

- **Modifying, Inserting, and Deleting a Record**

  This is handled by the form window. The form window also keeps track of whether any changes have been made to the current record, by setting a boolean variable bFormDirty, and can cancel any changes made.

- **Exiting from the Application**

  If a user chooses Exit from the File menu or presses the Exit push button on the toolbar, a SAM_Close message is sent to the MDI window. Upon receiving SAM_Close, the MDI window sends the SAM_Close message to the form window. The form window determines if there are any changes made by the user to the current form which has not been saved. If yes, it asks the user if the user wishes to lose these changes. The form window returns the final decision to the MDI window which takes the action to close the application or keep it alive.

Now let me show you the details of the implementation. I explain only the concepts that were not explained in earlier chapters.

Application Global Declarations

In this section, I discuss the global declarations of the application. Global declarations define the external functions I use from a Microsoft Windows system dynamic link library (DLL) called USER.EXE, user constants, named menus, global variables, class definitions, and application actions for SAM_AppStartup, SAM_AppExit, and SAM_SqlError.

External Functions

External functions are written in C or assembly language and are included in Windows (or OS/2) dynamic link libraries. You declare external functions in the Global Declarations External Functions section of the application outline. In this application, I use GetSystemMenu, EnableMenuItem, and ShowWindow from

the Microsoft Windows DLL called USER.EXE. First, let me show you how I have declared these functions.

```
Application Description:
  PHAD.APP
  MDI Windows
  Chapter 5
  Power Programming with SQLWindows
  by Rajesh Lalwani.
  Copyright (c) 1994 by Gupta Corporation.
  All rights reserved.
 Libraries
  File Include: SQLHANDL.APL
  File Include: REPORT.APL
 Global Declarations
  External Functions
   Library name: USER.EXE
    Function: GetSystemMenu
     Export Ordinal: 0
     Returns
      Number: WORD
     Parameters
      Window Handle: HWND
      Boolean: BOOL
    Function: EnableMenuItem
     Export Ordinal: 0
     Returns
      Boolean: BOOL
     Parameters
      Number: WORD
      Number: WORD
      Number: WORD
    Function: ShowWindow
     Description:
       The ShowWindow function sets the given window's
       visibility state.
     Export Ordinal: 0
     Returns
      Boolean: BOOL
     Parameters
      Window Handle: HWND
      Number: INT
```

Listing 5.2 External Functions used in PHAD.APP

Dynamic Link Library (DLL)

A DLL is a library file that contains external functions that Microsoft Windows applications can call to perform tasks. A DLL gives an application access to functions that are not part of its executable code. Although a DLL contains executable code, you cannot run a DLL as a program. Instead, an application loads a DLL and executes the functions in the DLL by linking to it dynamically at runtime. On the other hand, if an application is linked to a library using a linker, it is called static linking. The library is called a static library. With static linking, you combine the code for called functions with the application code when you create the application, but with a DLL you do not combine the code.

There are two main advantages of using DLLs. You can change a DLL's functions without changing the calling application as long as the names and definitions of the functions remain the same. If you have distributed your application to end users in the form of an .EXE file and DLLs, sending a patch to fix certain bugs may consist of sending just a new DLL. A DLL saves memory when two or more applications that use a common set of functions are running at the same time. Once a DLL is loaded, the system shares the DLL with any other program that needs it. Only one copy of the library is ever loaded at any time.

These functions must use the FAR PASCAL calling convention and they must be EXPORTED from the library (using ordinal numbers). To write a DLL, you must be competent in writing C, C++, or assembler programs. To call a DLL function from a SQLWindows application, you only need to know how to code for SQLWindows.

When to Use DLLs

Normally, you do not need to use DLLs while developing a SQLWindows application. However, you may want to use DLLs for:

- Using custom controls, Visual Basic Controls (VBX), or device drivers for special hardware.
- Using third-party libraries such as the one for sending e-mail from within a SQLWindows application.
- Accessing Microsoft Windows API functions in KERNEL.EXE, GDI.EXE, and USER.EXE to complement SAL functions provided by SQLWindows. In this

application, I have used three functions GetSystemMenu, EnableMenuItem, and ShowWindow from USER.EXE.

- Passing data between applications.

Declaring External Functions

As you can see in Listing 5.2, you declare an external function under Global Declarations. You specify the name(s) of the DLL as the Library Name. For each function in the DLL that the application calls, specify the following items:

- The name of the function. If you are specifying the ordinal number of the function, you can choose any name for the function.

- A description of what the function does. It is optional.

- The ordinal number of the function. Each function has an ordinal number declared in the library's .DEF file. Beginning with Release 4.1 of SQLWindows, you do not need to specify an ordinal number when you define an external function. Instead, you can spell the function name exactly as it is declared in the DLL and specify 0 (default) for the ordinal number. SQLWindows looks for the function name in the DLL and displays an error message if it cannot find the DLL or the function.

  You can still specify the ordinal number. If you do, you can give the function any name.

  You can determine a function's export ordinal number using tools such as MAPDLL.EXE and EXEHDR.EXE. MAPDLL.EXE comes with SQLWindows whereas EXEHDR.EXE comes with Microsoft compilers.

- The internal data type and the external data type of the return value if the function returns a value.

- The internal data type and the external data type of each parameter of the function, if any.

When you specify a data type for the return value and parameters of an external function, the data type has two parts separated by a colon, as seen in Listing 5.2. The first part is the internal data type. The second part is the external data type. The internal data type refers to the way SQLWindows would look at this return value or the parameter. The external data type refers to the way the external

function would look at this return value or the parameter. For example, an external function may have three parameters of data type BYTE, WORD, and FLOAT. But as far as a SQLWindows application is concerned, they are all of data type Number. SQLWindows uses the external data type to allocate bytes on the stack when the application calls the external function. You can look at the outline options bar or the manual for a complete list of external data types.

Beginning with Release 4.1 of SQLWindows, it is easier to call an external function with a C struct parameter. You can use the new data type structPointer to identify the elements in a C struct when you call external functions. This data type makes SWCSTRUC.DLL obsolete. (Even before Release 4.1, SQLWindows applications could pass data to and receive data from C structures in DLLs. SWCSTRUC.DLL provided functions to do this.) When you declare an external function parameter with the structPointer data type, SQLWindows bundles the items under it in a C struct and passes a pointer to it to the external function.

Calling External Functions

You call an external function the same way you call a SQLWindows (system or internal) function. You use a Set or Call statement or you embed it in an expression. Later in this chapter I explain each of the three functions I have declared as external functions.

User Constants—Defining Programmer Messages

As you have seen from the architecture of PHAD.APP, I make extensive use of messages for communication among the MDI window, the form window, and the table window. I also used some Microsoft Windows API functions as external functions which use some Microsoft Windows constants. I have defined the programmer messages and the Microsoft Windows constants as follows:

```
Constants
 System
 User
  Number: PM_GoToFirst = SAM_User
  Number: PM_GoToPrev = SAM_User+1
  Number: PM_GoToNext = SAM_User+2
  Number: PM_GoToLast = SAM_User+3
  Number: PM_GoToLParam = SAM_User+4
  Number: PM_Refresh = SAM_User+5
  Number: PM_Update = SAM_User+6
```

```
Number: PM_New = SAM_User+7
Number: PM_Delete = SAM_User+8
Number: PM_Undo = SAM_User+9
Number: PM_IndexLetter = SAM_User+10
Number: PM_Initialize = SAM_User+11
Number: PM_Dirty = SAM_User+12
Number: MAX_ZOOMIN = 1024
! Microsoft Windows constants
Number: MF_BYPOSITION = 0x0400
Number: MF_DISABLED = 0x0002
Number: MF_GRAYED = 0x0001
Number: SW_MINIMIZE = 6
Number: SW_SHOWNORMAL = 1
! SQL Error handling constants
Number: ERROR_TIMEOUT = -1805
Number: ERROR_ROWID = 806
```

Listing 5.3 Programmer messages, Microsoft Windows constants, etc. defined as user constants.

SAM_User is a system constant. It represents the value at which user-defined messages should start. The next consecutive user-defined message would be SAM_User+1. The value of SAM_User is 0x4000.

MF_BYPOSITION, MF_DISABLED, etc. are Microsoft Windows constants which I need while calling the external functions from USER.EXE. Since these are Microsoft Windows constants, I need to define them in my SQLWindows application. While it is not necessary to use exactly the same names for the constants as Microsoft Windows SDK (Software Development Kit), it is a good practice. This results in less confusion and ease of maintenance if someone else maintains this code down the road. I got the values of these constants (such as 0x0001 for MF_GRAYED) from the WINDOWS.H file that came with Microsoft SDK.

Named Menus

You can define a menu in an MDI window but it is only active when the currently active child window does not define its own menu. When an MDI child window is active and the child window has defined its own menu, its menu replaces the MDI window's menu in the MDI window's menu bar.

You can use named menus to create popup menus that different windows can share. You can define a named menu in the Named Menus sections of Global

Declarations, the MDI window or other windows such as a form window, or a table window. You can only use named menus for top-level popup menus, not for cascading menus in a popup menu.

In PHAD.APP, I have three popup menus at all times – File, Edit, and Window. In addition, I have another popup menu Record whenever the form window is active. I chose to implement File, Edit, and Window as named menus which are used by both the form window and the table window. The form window defines Record popup menu on its own.

Predefined Named Menus

NEWAPP.APP, the default starter application, contains these predefined named menus:

- menuEdit—An Edit menu with Undo, Cut, Copy, Paste, and Clear.
- menuOLEEdit—The same as menuEdit, but with OLE menu items for picture windows.
- menuMDIWindows—An MDI window's menu with commands to manage MDI child windows.

Listing 5.4 shows the named menus I have used in PHAD.APP.

```
Named Menus
 Menu: menuEdit
 Windows Menu: menuMDIWindows
 Menu: menuFile
   Title: &File
   Description: Menu for Exiting from the application
     and displaying the about dialog box.
   Status Text: Exit and About menu items
   Menu Item: E&xit
    Status Text: Exit from this application
    Menu Actions
     ! Post SAM_Close message to the MDI window
     Call SalPostMsg(hWndMDI,SAM_Close,0,0 )
   Menu Separator
   Menu Item: &About
    Status Text: About PhAd
```

```
Menu Actions
  ! Display the about dialog box
  Call SalModalDialog( dlgAbout, hWndForm )
```

Listing 5.4 PHAD.APP uses two predefined named menus (menuEdit and menuMDIWindows) and defines a new named menu menuFile .

Global Variables

I define a global sql handle, hSqlTable, to be used during the initial login process. I define some other variables to store the sql handle causing the SQL error, the error number, the position of the error within the SQL statement, the error message, and whether the transaction has been rolled back. These variables are used by the function HandleSQLError.

```
Variables
  ! Variables used by first connect
  Boolean: bConnectTable
  Sql Handle: hSqlTable
  ! Variables used by global error handler
  Sql Handle: hSqlError
  Number: nError
  Number: nPos
  String: strMessage
  Boolean: bRollback
```

Listing 5.5 Variables defined by the application's Global Declarations section.

Class Definitions

Listing 5.6 shows the classes used by PHAD.APP. clsSqlHandle and clsSqlHandleSelect are defined in SQLHANDL.APL and are the same as those used in Chapter 4. clsdlgReport is a new dialog box class, which I define in REPORT.APL. Internal function HandlSQLError is used by the Error function of clsSqlHandlePhAd and clsSqlHandleSelectPhAd and the global SQL error handler.

```
Internal Functions
 Function: HandleSQLError
  Description: This function displays the app message,text,
  cause, and remedy in a dialog box.
```

```
Actions
  ! Get the Sql Handle, error number and position of the last
    error.
  Call SqlExtractArgs
    ( wParam, lParam, hSqlError, nError, nPos )
  ! See if the system initiated any rollback.hSqlError would be
  ! invalid if the error occurred during first SqlConnect.
  ! 405 Invalid user name
  ! 404 Invalid Password
  ! 401 Cannot open database
  If nError != 405
    and nError != 404
    and nError != 401
    and hSqlError != hWndNULL
   Call SqlGetRollbackFlag( hSqlError, bRollback )
   If not bRollback
    ! Rollback the transaction so that we release any locks
    ! held before we display the dialog box.
    Call SqlImmediate ('ROLLBACK')
    Set bRollback = TRUE
   Set strMessage = 'The transaction has been rolledback. '
   Select Case nError
    Case ERROR_ROWID
     Set strMessage = strMessage ||
       'This record has been updated since you fetched it.
        Please Refresh and try again.'
     Break
    Case ERROR_TIMEOUT
     Set strMessage = strMessage ||
       'Timed out waiting for a lock. Please try again.
       You will have to Refresh after that.'
     Break
    Default
     Set strMessage = strMessage || 'Please Refresh.'
  Else
   Set strMessage = 'Enter the correct values and try again.'
  Call SalModalDialog
   ( dlgSqlError, hWndNULL, nError, strMessage )
Class Definitions
 Pushbutton Class: clsIndexPushButton
  Title: A
  Picture Transparent Color: Gray
  Background Color: White
  List in Tool Palette? Yes
  Description:
```

```
    This push button class sends a message to the mdi window
    that the letter corresponding to the instance variable
    'nIndexLetter' has been pressed.
  Instance Variables
   Number: nIndexLetter
  Functions
   Function: Initialize
    Description: This function initializes the instance variable
      nIndexLetter
    Parameters
     String: strLetter
    Actions
     Set nIndexLetter = SalStrLop( strLetter )
  Message Actions
   On SAM_Click
    Call SalPostMsg( hWndMDI, PM_IndexLetter, nIndexLetter, 0 )
General Window Class: clsGenResetDirty
  Description: This general window class blanks itself
    upon receiving PM_Initialze message. Also,
    upon SAM_Validate, it sets bFormDirty to
    TRUE by posting a PM_Dirty message to hWndForm.
  Message Actions
   On SAM_AnyEdit
    Call SalSendMsg( hWndForm, PM_Dirty, 0, 0 )
   On PM_Initialize
    Call SalClearField( hWndItem )
Data Field Class: clsDfResetDirty
  Description:
    This data field class initializes itself
    upon receiving PM_Initialze message. Also,
    upon SAM_Validate, it sets bFormDirty to
    TRUE.
  Derived From
   Class: clsGenResetDirty
Multiline Field Class: clsMlResetDirty
  Description:
    Same as clsDfResetDirty except that it is
    a multi-line field class.
  Derived From
   Class: clsGenResetDirty
Functional Class: clsSqlHandleSelectPhAd
  Description:
    This class is derived from clsSqlHandleSelect
    defined in Chapter 4 (SQLHANDL.APL). It
    defines new member functions SetParameter,
```

```
    and GetResultSetCount, and overrides Error.
  Derived From
   Class: clsSqlHandleSelect
  Functions
   Function: SetParameter
    Description:
      This function is used to set a database
      parameter such as DBP_FETCHTHROUGH or
      DBP_PRESERVE.
    Returns
     Boolean:
    Parameters
     ! The database parameter to be set
     Number: nParameterParm
     ! Number value of the parameter
     Number: nSetting
     ! String value of the parameter
     String: strSetting
    Actions
     Return SqlSetParameter(
       sqlHandle, nParameterParm, nSetting, strSetting )
   Function: GetResultSetCount
    Description:
      This function simply returns the
      instance variable nResultSetCount.
    Returns
     Number:
    Actions
     Return nResultSetCount
   Function: Error
    Description: This function overrides the function
      Error of clsSqlHandle.  This function
      displays the error number of the last
      error, error text, cause and remedy.
    Parameters
     String: strMessage
    Actions
     ! HandleSQLError function extracts the SQL Error args
       from wParam and lParam.
     Call HandleSQLError(  )
 Functional Class: clsSqlHandlePhAd
  Description: This class is derived from clsSqlHandle
    defined in Chapter 4 (SQLHANDL.APL). It
    overrides Error.
```

```
Derived From
 Class: clsSqlHandle
Functions
 Function: Error
  Description: This function overrides the function
    Error of clsSqlHandle.  This function
    displays the error number of the last
    error, error text, cause and remedy.
  Parameters
   String: strMessage
  Actions
   ! HandleSQLError function extracts the SQL Error args
     from wParam and lParam.
   Call HandleSQLError(  )
Functional Class: clsSqlHandle
Functional Class: clsSqlHandleSelect
Dialog Box Class: clsdlgReport
```

Listing 5.6 Classes used by PHAD.APP. clsSqlHandle and clsSqlHandleSelect are defined in SQLHANDL.APL. clsdlgReport is defined in REPORT.APL.

clsIndexPushButton

As you can see in Figure 5.1, PhAd provides index push buttons on the toolbar for letters A through Z. When a user presses a button, for example, A, the push button sends a message PM_IndexLetter to the MDI window with wParam containing the ASCII value of A. Instead of repeating the code 26 times, I chose to define a push button class clsIndexPushButton to contain the common code. clsIndexPushButton has an instance variable called nIndexLetter which contains the ASCII value of the letter that the instance of this class represents. To initialize nIndexLetter, each instance calls the Initialize member function at the time of creation. I remember ASCII value (number) of the letter, because it is easier to compare two numbers during binary search.

Initialize has one string parameter. The Initialize function uses the SalStrLop function to extract the ASCII numeric value of the first character of this string parameter in decimal format. SalStrLop function returns zero (0) when the string is null. This function removes the first character of the string. Since I have defined strLetter as a String and not as a Receive String, calling this function does not affect the string of the caller.

When an instance of this class gets SAM_Click, the class posts a PM_IndexLetter message to the MDI window. hWndMDI is a system variable. This variable is the window handle of the current MDI frame. Note that the class also sends the value of nIndexLetter in wParam so that the MDI window would know which letter is desired by the user.

General Window Class

Beginning with Release 5.0 of SQLWindows, you can define a general window class such as clsGenResetDirty in Listing 5.6. You can put the common code here in this general window class and derive other specific window classes from it.

clsGenResetDirty responds to PM_Initialize message by clearing the field by calling the SalClearField function. Note the use of the special variable hWndItem. clsGenResetDirty also processes SAM_AnyEdit and sends a PM_Dirty message to the form window so that the form window can set the boolean variable bFormDirty.

As discussed in Chapter 3, I do not use SAM_Validate because SQLWindows does not send a SAM_Validate message if the user performs some action such as going to the next record by using the *menu*.

clsDfResetDirty and clsMlResetDirty

I define these classes so that when I derive instances of these data field and multi-line field classes, the instances respond to a PM_Initialize message; and send PM_Dirty message to the form window when their value has changed. Keeping the common code in the general window class clsGenResetDirty avoids duplicating the code in both clsDfResetDirty and clsMlResetDirty.

clsSqlHandleSelectPhAd

I define a new functional class, clsSqlHandleSelectPhAd, which is derived from clsSqlHandleSelect of SQLHANDL.APL. This is a very good example of using OOP for code reusability. Basically, I used everything clsSqlHandleSelect offers plus a little more. By deriving clsSqlHandleSelectPhAd from clsSqlHandleSelect, I got everything that the latter had. In addition, I defined two new functions: SetParameter, and GetResultSetCount. Also, I chose to override the function Error.

I need access to the instance variable sqlHandle to set database parameters by using SqlSetParameter. To control the enabling and disabling of push buttons I need access to another instance variable nResultSetCount. In the true spirit of OOP, I do not want to access these instance variables directly. That is why I defined two functions—SetParameter, and GetResultSetCount.

clsSqlHandleSelect (actually, clsSqlHandle) provides very little information in case of a SQL error. I have chosen to override the Error function by defining it again in clsSqlHandleSelectPhAd. This function calls HandleSQLError to display the error number, application generated error message, error text, reason, and remedy in a dialog box dlgSqlError. This dialog box is essentially the same as the dlgSqlError of FRMBROWS.APL (Chapter 4) except that it has a slightly different user interface. I explain the differences later on in this chapter.

clsSqlHandlePhAd

I define a new functional class, clsSqlHandlePhAd, which is derived from clsSqlHandle of SQLHANDL.APL. I define this new class to override the Error function to provide more information in case of a SQL error.

Application Actions

I process SAM_AppStartup to present a login dialog box to the user. If everything goes well, I call SqlSetIsolationLevel to set the isolation level to 'RL' and SalCreateWindow to create the MDI window mdiWindow. The child windows mdifrmOnePerson and mditblTable have the 'Automatically Create' attribute set to Yes, so they are created also.

Upon receiving SAM_AppExit, I disconnect hSqlTable. Since the application connects it during SAM_AppStartup, I chose to disconnect it here as opposed to the SAM_Destroy section of the table window.

In case of a SQL error, the global error handler (On SAM_SqlError) calls HandleSQLError to display the error number, application error message, error text, reason, and remedy.

```
Application Actions
 On SAM_AppStartup
  ! Login Dialog Box with defaults for database,
    user, password. Pass hSqlTable as the receive
    parameter.
  Set bConnectTable = SalModalDialog( dlgLogin, hWndNULL,
```

```
      'PHAD', 'SYSADM', 'SYSADM', hSqlTable )
    ! Quit the application if connect not successful
    If not bConnectTable
     Call SalQuit()
    ! Connect successful, set isolation level to 'RL'
      and create the MDI window
    Call SqlSetIsolationLevel( hSqlTable, 'RL' )
    Call SalCreateWindow( mdiWindow, hWndNULL )
  On SAM_AppExit
    ! Disconnect the sql handle of our first connect
    If bConnectTable
     Call SqlDisconnect( hSqlTable )
  On SAM_SqlError
    Call HandleSQLError()
    ! Return FALSE so that the Sql function which caused the error
      would return FALSE to its caller.
    Return FALSE
```

Listing 5.7 The Application Actions section of PHAD.APP.

Dialog Box to Report a SQL Error

In Chapter 3, I defined dlgSqlError to report a SQL error. In this chapter, I modify this dialog box to change the user interface. Now, dlgSqlError displays the SQL error number in a data field. There is only one multi-line field to display the application generated error message, error text, reason, and remedy. The user can switch among them using radio buttons on the top. The user can choose to continue, in which case the dialog box ends and returns FALSE, or halt the application. Figure 5.2 shows how the dialog box looks in case a user types an incorrect user name during the login process. Listing 5.8 shows the code. Except for using radio buttons, it is very similar to dlgSqlError defined in Chapter 3.

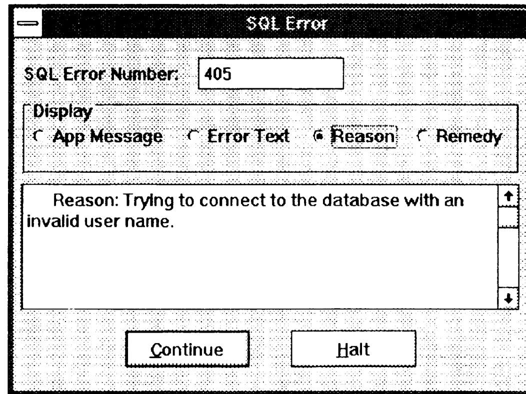

Figure 5.2 dlgSqlError displays a SQL error. In this case the user typed an incorrect user name.

```
Dialog Box: dlgSqlError
 Title: SQL Error
 Description:
   This dialog box displays the application
   message, error text,
   cause and remedy corresponding to the
   SQL error number in the first data field.
   Initially, this data field contains nError,
   but it can be changed.
 Contents
  Data Field: dfSQLErrorNum
   Data
    Data Type: Number
    Editable? Yes
   Message Actions
    On SAM_Create
     Set dfSQLErrorNum = nError
  Background Text: SQL Error Number:
  Group Box: Display
  Radio Button: rbAppMsg
   Title: App Message
   Message Actions
    On SAM_Click
     Set mlText = strMessage
  Radio Button: rbErrorText
   Title: Error Text
```

```
    Message Actions
     On SAM_Click
      Set mlText = SqlGetErrorTextX( dfSQLErrorNum )
   Radio Button: rbReason
    Title: Reason
    Message Actions
     On SAM_Click
      Call SqlErrorText( dfSQLErrorNum, SQLERROR_Reason,
        mlText, SalGetMaxDataLength( mlText ), nMaxCause )
   Radio Button: rbRemedy
    Title: Remedy
    Message Actions
     On SAM_Click
      Call SqlErrorText( dfSQLErrorNum, SQLERROR_Remedy,
        mlText, SalGetMaxDataLength( mlText ), nMaxCause )
   Multiline Field: mlText
    Data
     Maximum Data Length: 1024
     String Type: Long String
     Editable? No
    Display Settings
     Word Wrap? Yes
     Vertical Scroll? Yes
   Pushbutton: pbContinue
    Title: &Continue
    Keyboard Accelerator: Enter
    Message Actions
     On SAM_Click
      Call SalEndDialog( hWndForm, FALSE )
   Pushbutton: pbHalt
    Title: &Halt
    Message Actions
     On SAM_Click
      Call SalQuit()
  Window Parameters
   Number: nError
   String: strMessage
  Window Variables
   Number: nMaxCause
   Number: nMaxRemedy
  Message Actions
   On SAM_Create
    ! Since we come here during the execution
      of a SQL operation, cursor might be a wait cursor.
      Show the normal cursor.
```

```
Call SalWaitCursor( FALSE )
! The first text to be displayed should be
  the Error Text
Call SalPostMsg( rbErrorText, SAM_Click, 0, 0 )
Set rbErrorText = TRUE
```

Listing 5.8 Dialog box dlgSqlError to display SQL error number, application generated message, error text, reason, and remedy.

Radio Buttons

As you can see in Figure 5.2, I used a group of three radio buttons. You use a group of radio buttons for mutually exclusive options. Only one radio button in a group can be On at a time. Clicking on a radio button sets that button On and turns Off whichever button in the same group was previously On. If you click on a radio button that is already On, no SAM_Click message is sent and the radio button remains On. The user can also press the TAB key to move to a checked radio button in a group and then use the arrow keys to move the input focus to another radio button in the group.

Radio buttons evaluate to Boolean values. They can be either On (TRUE) or Off (FALSE).

When a collection of radio buttons is contiguous in the outline but you want to process them as two different groups, you can use a group box to separate them. A group separator can also be used to separate contiguous but unrelated groups of radio buttons. The Group Separator item is listed in the Outline Options dialog box when the contents section of a form window or dialog box is highlighted in design mode.

MDI Window

Listing 5.9 shows the relevant portions of code for the MDI window mdiWindow. It shows the toolbar, contents, functions, variables, and message actions. The overview is in an earlier section on the architecture of this application. Most of the code is very straightforward and easy to follow. The next few sections explain the new concepts used.

```
MDI Window: mdiWindow
  Title: PhAd
  Display Settings
```

```
 Automatically Created at Runtime? No
Icon File: PHAD.ICO
Accesories Enabled? Yes
Description:
  This is an MDI window for the application.
  It acts like a glue to hold the child windows
  together. Most of the actual work is done
  by child windows.
Toolbar
 Contents
  Pushbutton: pbFirst
   Title: First
   Picture File Name: FIRST.BMP
   Message Actions
    On SAM_Click
     If mdifrmOnePerson.OkToLoseChangesIfAny()
      Call SalSendMsgToChildren
        (hWndMDI, PM_GoToFirst, 0, 0 )
  Pushbutton: pbPrev
   Title: Prev
   Picture File Name: PREV.BMP
   Message Actions
    On SAM_Click
     If mdifrmOnePerson.OkToLoseChangesIfAny()
      Call SalSendMsgToChildren
        (hWndMDI, PM_GoToPrev, 0, 0 )
  Pushbutton: pbNext
   Title: Next
   Picture File Name: NEXT.BMP
   Message Actions
    On SAM_Click
     If mdifrmOnePerson.OkToLoseChangesIfAny()
      Call SalSendMsgToChildren
        (hWndMDI, PM_GoToNext, 0, 0 )
  Pushbutton: pbLast
   Title: Last
   Picture File Name: LAST.BMP
   Message Actions
    On SAM_Click
     If mdifrmOnePerson.OkToLoseChangesIfAny()
      Call SalSendMsgToChildren
        (hWndMDI, PM_GoToLast, 0, 0 )
  Pushbutton: pbZoomIn
   Title: &ZoomIn
   Picture File Name: ZOOMIN.BMP
```

```
 Message Actions
  On SAM_Click
   If bFormActivated
    ! At the moment, the form is activated
    Set hWndFocus = SalGetFocus(  )
    If SalGetWindowText
      ( hWndFocus, strZoomIn, MAX_ZOOMIN ) < 0
     Set strZoomIn =
       '**ERROR** Could not get window text.'
   Else
    ! At the moment, the table window is activated
    Call mditblTable.GetFocusRowStr(strZoomIn)
   Call SalModalDialog( dlgZoomIn, hWndMDI, strZoomIn )
Pushbutton: pbRefresh
 Title: &Refresh
 Picture File Name: FETCH.BMP
 Message Actions
  On SAM_Click
   Call SalPostMsg( hWndMDI, PM_Refresh, 0, 0 )
Pushbutton: pbForm
 Title: &Form
 Picture File Name: FORM.BMP
 Message Actions
  On SAM_Click
   ! display the form window in normal mode
     if it is minimized at present. If it is
     not minimized, simply bring it to top.
   If SalGetWindowState( mdifrmOnePerson ) =
     Window_Minimized
    Call ShowWindow( mdifrmOnePerson, SW_SHOWNORMAL )
   Else
    Call SalBringWindowToTop( mdifrmOnePerson )
Pushbutton: pbTable
 Title: &Table
 Picture File Name: TABLE.BMP
 Message Actions
  On SAM_Click
   ! display the table window in normal mode
     if it is minimized at present. If it is
     not minimized, simply bring it to top.
   If SalGetWindowState( mditblTable ) =
     Window_Minimized
    Call ShowWindow( mditblTable, SW_SHOWNORMAL )
   Else
    Call SalBringWindowToTop( mditblTable )
```

```
    Pushbutton: pbPrint
     Title: &Print
     Picture File Name: PRINT.BMP
     Message Actions
      On SAM_Click
       Call SalModalDialog( dlgReportViewPrint, hWndMDI )
    Pushbutton: pbExit
     Title: E&xit
     Picture File Name: EXIT.BMP
     Message Actions
      On SAM_Click
       Call SalPostMsg(hWndMDI,SAM_Close,0, 0 )
    Line
    Pushbutton: pbA
     Class: clsIndexPushButton
     Title: A
     Message Actions
      On SAM_Create
       Call pbA.Initialize('A')
    Pushbutton: pbZ
     Class: clsIndexPushButton
     Title: Z
     Message Actions
      On SAM_Create
       Call pbZ.Initialize('Z')
  Contents
   Form Window: mdifrmOnePerson
   Table Window: mditblTable
  Functions
   Function: BeginOperation
    Description: This function puts the wait
      cursor and sets the MDI status text
    Parameters
     String: strText
    Actions
     Call SalWaitCursor( TRUE )
     Call SalStatusSetText(hWndMDI, strText )
   Function: EndOperation
    Description: This function removes the wait
      cursor and resets the MDI status text
    Actions
     Call SalWaitCursor( FALSE )
     Call SalStatusSetText(hWndMDI, '' )
   Function: DisableClose
    Description:
```

```
      This function will disable the Close
      menu item from the system menu of hWndParm
    Parameters
     Window Handle: hWndParm
    Local variables
     Number: hSystemMenu
    Actions
     ! Call GetSystemMenu to get menu handle of a copy
       (fRevert = FALSE).
     Set hSystemMenu = GetSystemMenu (hWndParm, FALSE)
     ! Disable Close which is menu item 6 (0-based) by position.
     Call EnableMenuItem(hSystemMenu, 6,
       MF_BYPOSITION | MF_DISABLED | MF_GRAYED)
  Window Variables
   Boolean: bReturn  ! boolean return
   Number: nReturn  ! number return
   Window Handle: hWndFocus
   Long String: strZoomIn
   Boolean: bFormActivated
  Message Actions
   On SAM_Create
    ! Post PM_Refresh message to the MDI window
    Call SalPostMsg( hWndMDI, PM_Refresh, 0, 0 )
   On PM_Refresh
    If mdifrmOnePerson.OkToLoseChangesIfAny()
     ! Send PM_Refresh message to all the children
       of MDI Window
     Set bReturn = SalSendMsgToChildren
       (hWndMDI, PM_Refresh, wParam, lParam )
   On PM_GoToLParam
    If mdifrmOnePerson.OkToLoseChangesIfAny()
     Set bReturn = SalSendMsgToChildren
       (hWndMDI, PM_GoToLParam, wParam, lParam)
   On PM_IndexLetter
    Call BeginOperation( 'Searching...' )
    Call SalPostMsg( hWndMDI, PM_GoToLParam, 0,
      SalSendMsg( mditblTable, PM_IndexLetter, wParam, lParam ) )
    Call EndOperation(  )
   On SAM_Close
    ! Ask the form if ok to exit
    If not SalSendMsg( mdifrmOnePerson, SAM_Close, 0, 0 )
     Return FALSE
```

Listing 5.9 MDI window, toolbar, contents, functions, variables, and message actions.

SalSendMsgToChildren

SalSendMsgToChildren sends a message to all child items of a form window, dialog box, table window, or MDI window. It returns TRUE if the function succeeds, FALSE otherwise. I used this function to broadcast a certain message to all the children. In this case, the form window and the table window. Later, I have use this function to send a PM_Initialize message to all the child objects on the form window so that each individual object (data fields and multi-line field) receives this message and resets itself to a blank. Using SalSendMsgToChildren results in clean, easy to understand code and requires less maintenance down the road as more child objects (such as data fields) are added to the form window.

pbZoomIn

A user can use the ZoomIn push button to see, in a dialog box, the contents of a field on the form window. pbZoomIn shows the contents of the field which has focus at the moment. If the table window has the focus, the dialog box shows the complete name of the person in the focus row (see Chapter 6).

Who has the focus?

First, pbZoomIn finds out which child window is active at the moment by looking at the boolean variable bFormActivated. This variable is maintained by the form window. If the form window is not active, pbZoomIn simply calls the GetFocusRowStr of the table window by qualifying the name of the function with the name of the table window mditblTable. I discuss the table window and GetFocusRowStr in Chapter 6.

On the other hand, if the form window is currently active, pbZoomIn first calls SalGetFocus to get the handle of the window with the focus. SalGetFocus returns the window handle of the window that can currently receive input from the keyboard.

pbZoomIn then calls SalGetWindowText to retrieve the text of the window. For a data field, multiline text field, or table window column, window text is the field value in string form, regardless of the object's data type. Window text is the title of a form window, dialog box, table window, radio button, check box, background text, or push button. SalGetWindowText returns a number that indicates the length of the text string. It is zero (0) if the window has no text.

pbZoomIn finally displays the string strZoomIn in a multi-line text field by creating a modal dialog box from the dlgZoomIn template.

Mnemonics

Notice that the title of the push button is &ZoomIn. Using an ampersand character (&) defines the character following & as a mnemonic. In this case Z would be a mnemonic and the title of the push button appears as ZoomIn.

When the user presses a mnemonic key at the same time as the Alt key, the input focus moves to the object. You can assign a mnemonic to background text, columns, radio buttons, group boxes, and check boxes.

You can also use mnemonics with menu items and push buttons. A mnemonic for a menu item and a push button does more than move the input focus. For a menu item, it also invokes the menu item's actions. For a push button, it sends a SAM_Click message to the push button.

You can create a mnemonic by typing an ampersand character (&) before the character in the object's title that you want as a mnemonic. A mnemonic character appears in an object's title with an underscore. To add an ampersand character itself, use two ampersands together (&&).

You can give a label (name) to a data field using background text. A background text mnemonic is associated with a data field to move the input focus to that data field. For example, if the background text for the data field dfLastName is &Last Name, pressing Alt-L moves the focus to dfLastName. The background text must come immediately before the data field in the outline; the background text can appear visually anywhere in the form window or dialog box.

Manipulating a Child Window's Visibility State

The push button pbForm is used to restore the form window if it is minimized at the moment. If the form window is not minimized, pbForm brings it to the top in case it is not the active child window at the moment. pbTable does the same for the table window. To accomplish this, both pbForm and pbTable make use of two SAL functions (SalGetWindowState and SalBringWindowToTop) and one Microsoft Windows API function (ShowWindow).

SalGetWindowState

This function returns a window's current state. The return value is a constant that indicates a window's current state. It is equal to one of a predefined set of values.

Constant	Description
Window_Invalid	Specified window handle is not valid.
Window_Maximized	Specified window is maximized. You can also specify this constant when you call the SalLoadAppAndWait function.
Window_Minimized	Specified window is minimized (iconic). You can also specify this constant when you call the SalLoadAppAndWait function.
Window_Normal	Specified window exists on the screen and is neither maximized nor minimized. You can also specify this constant when you call the SalLoadAppAndWait function.
Window_NotVisible	Specified window is not visible on the screen. You can also specify this constant when you call the SalLoadAppAndWait function.

SalBringWindowToTop

This function brings a window to the top of all overlapping windows. It returns TRUE if the function succeeds, FALSE otherwise.

ShowWindow

ShowWindow is a Microsoft Windows API function that sets a window's visibility state. The return value is nonzero if the window was previously visible. It is zero if the window was previously hidden. The first parameter specifies the handle of the window. The second parameter specifies how the window is to be shown. This parameter can be one of the following values:

Constant	Numeric Value	Description
SW_HIDE	0	Hides the window and passes activation to another window.
SW_MINIMIZE	6	Minimizes the specified window and activates the top-level window in the system's list.
SW_RESTORE	9	Activates and displays a window. If the window is minimized or maximized, Windows restores it to its original size and position (same as SW_SHOWNORMAL).
SW_SHOW	5	Activates a window and displays it in its current size and position.
SW_SHOWMAXIMIZED	3	Activates a window and displays it as a maximized window.
SW_SHOWMINIMIZED	2	Activates a window and displays it as an icon.
SW_SHOWMINNOACTIVE	7	Displays a window as an icon. The window that is currently active remains active.
SW_SHOWNA	8	Displays a window in its current state. The window that is currently active remains active.
SW_SHOWNOACTIVATE	4	Displays a window in its most recent size and position. The window that is currently active remains active.

Constant	Numeric Value	Description
SW_SHOWNORMAL	1	Activates and displays a window. If the window is minimized or maximized, Windows restores it to its original size and position (same as SW_RESTORE)

Index Push Buttons

In Listing 5.9, I have reproduced the code for two push buttons pbA and pbZ. The code for the other 24 push buttons is similar. Notice that each of these buttons call Initialize function to initialize the instance variable nIndexLetter at the time of creation.

PM_IndexLetter

Upon receiving a PM_IndexLetter, the MDI window calls SalSendMsg to send a PM_IndexLetter message to the table window. Remember that the wParam contains the ASCII value of the index letter that the user wants. In Chapter 6, I show you how the table window searches for this letter in the Last Name column. If it succeeds in finding a row which has a last name starting with this letter, it returns its row number. If it does not succeed, it returns the row number of the row where such a last name can be inserted. Note that SalSendMsg does not return until the processing for the message is complete. Also, the return value of SalSendMsg is what is actually returned by the table window's message handler for PM_IndexLetter. The MDI window uses this row number in lParam to post a PM_GoToLParam message to all its children.

SAM_Close

When the MDI window receives SAM_Close, it sends a SAM_Close message to the form window to ask if it is ok to exit from the application. If the form window returns FALSE, the MDI window returns FALSE and the MDI window does not close. Otherwise, the MDI window does not execute a return statement and the MDI window closes.

Functions

The MDI window defines three functions—BeginOperation, EndOperation, and DisableClose. BeginOperation calls SalWaitCursor to display the wait cursor. It also calls SalStatusSetText to display the specified text in the status bar of the MDI window. SalStatusSetText is used to display specified text in the status bar of a top-level or MDI window.

EndOperation displays the normal cursor again and removes the text from the status bar by displaying an empty string.

Accessing System Menu

Since both the form window and the table window are so important and interdependent, I did not want a user to be able to close either of them. As you can see in the code for the form window and table window, upon receiving SAM_Create, they both call the DisableClose function to disable the Close menu item in their System menu. System menu is the menu represented by a '—' symbol in the upper left corner of the window. To accomplish this, DisableClose calls two Microsoft Windows functions—GetSystemMenu and EnableMenuItem.

GetSystemMenu

The GetSystemMenu function allows an application to access the System menu for copying and modification.

The first parameter identifies the window that owns a copy of the System menu. The second parameter, fRevert, specifies the action to be taken. If this parameter is FALSE, the GetSystemMenu function returns a handle of a copy of the System menu currently in use. This copy is initially identical to the System menu, but can be modified. If the parameter is TRUE, GetSystemMenu resets the System menu back to the Windows default state. The previous System menu, if any, is destroyed. The return value is undefined in this case.

The handle that GetSystemMenu returns can be used with the AppendMenu, InsertMenu, or ModifyMenu function to change the System menu. In this case, I have used it with EnableMenuItem to disable the Close menu item.

EnableMenuItem

The EnableMenuItem function enables, disables, or grays (dims) a menu item.

The first parameter identifies the menu. The second parameter specifies the menu item to be enabled, disabled, or grayed. This parameter can specify pop-up menu items as well as standard menu items. The interpretation of this parameter depends on the value of the third parameter. The third parameter specifies the action to take. This parameter can be MF_DISABLED, MF_ENABLED, or MF_GRAYED, combined with MF_BYCOMMAND or MF_BYPOSITION. These values have the following meanings:

Value	Numeric Value	Meaning
MF_BYCOMMAND	0x0000	Specifies that the second parameter gives the menu-item identifier.
MF_BYPOSITION	0x0400	Specifies that the second parameter gives the position of the menu item (the first item is at position zero).
MF_DISABLED	0x0002	Specifies that the menu item is disabled.
MF_ENABLED	0x0000	Specifies that the menu item is enabled.
MF_GRAYED	0x0001	Specifies that the menu item is grayed.

The return value is 0 if the menu item was previously disabled, 1 if the menu item was previously enabled, and -1 if the menu item does not exist.

Form Window

Listing 5.10 shows the relevant portions of the outline for the form window. Most of the code is strikingly similar to the code for DATABASE.APP of Chapter 3. One major difference is that in DATABASE.APP, operations such as going to the first, previous, next, and last records, inserting, deleting, updating a record, etc.

are performed by the SAM_Click message handlers of push buttons. In this application, however, the work is performed by message handlers for messages such as PM_GoToFirst, PM_GoToPrev, PM_New, PM_Delete, PM_Update, etc.

Notice the use of SalSendMsgToChildren and PM_Initialize. Upon receiving PM_New, the form window simply sends a PM_Initialize message to all its child objects (data fields, multi-line field). The base class clsGenResetDirty of these child objects responds to the PM_Initialize message by clearing their value.

Also, see how the form window processes the SAM_Activate message to maintain the boolean variable bFormActivated.

In Chapter 3, in DATABASE.APP, I processed SAM_AppExit to disconnect all the sql handles. In this application, since the sql handles (actually instances of clsSqlHandleSelectPhAd and clsSqlHandlePhAd classes) are defined as the variables of the form window, they must be disconnected by the form window itself. I do so upon receiving SAM_Destroy.

SAM_Destroy

SAM_Destroy is sent to a top-level window (dialog box, form window, or table window) and then to all of its children just before the windows are destroyed. SAM_Destroy messages are sent after SAM_Close has been sent to the top-level window. For example, if a form window has data fields, SQLWindows sends the messages to the objects in this order:

1. SAM_Close to the form window.
2. SAM_Destroy to the form window.
3. SAM_Destroy to each of the form window's child windows.

After all of the SAM_Destroy messages are sent, the top-level and child windows are destroyed.

SAM_Destroy is also sent to an MDI window.

Menu Item—Enabled When?

Notice the use of the 'Enabled when' attribute of menu items such as First, Next, etc. of the Record pop up menu. You place an expression here that controls when the menu item is enabled. When the expression is TRUE, the menu item is

enabled and the user can select it to invoke the associated action. When enabled, the menu item name is black.

When the expression is FALSE, the menu item is disabled and the user cannot select it to invoke the associated action. When disabled, the menu item name is gray.

For a top-level menu item, the expression is only evaluated when you call SalDrawMenuBar.

Since the push buttons on the MDI window's toolbar are already disabled and enabled by the form window, depending on where a user is in the result set, I decided to enable and disable these menu items based on the state of the push buttons. This way, the form doesn't have to do extra work for the menu items and everything is kept in synchronization.

Now, I will leave you with the relevant portions of outline for the form window. It is a rather long listing; I have included it for the sake of completeness. This way it will be easier to look at the code as compared to looking up the companion disk.

```
Form Window: mdifrmOnePerson
    Title: PhAd Form
    Accesories Enabled? No
    Display Settings
     Automatically Created at Runtime? Yes
     Resizable? No
    Description:
      This form window displays information about
      one person. It can also be used to modify
      this information, delete this record or
      insert a new record.
    Menu
     Named Menu: menuFile
     Named Menu: menuEdit
     Popup Menu: &Record
      Status Text: Browse through records, modify, insert,
        update, delete a record.
      Menu Item: &First
       Status Text: Go to the first record
       Menu Settings
        Enabled when: SalIsWindowEnabled( mdiWindow.pbFirst )
```

```
    Menu Actions
     Call SalPostMsg( mdiWindow.pbFirst, SAM_Click, 0, 0 )
   Menu Item: &Previous
    Status Text: Go to the previous record
    Menu Settings
     Enabled when: SalIsWindowEnabled( mdiWindow.pbPrev )
    Menu Actions
     Call SalPostMsg( mdiWindow.pbPrev, SAM_Click, 0, 0 )
   Menu Item: &Next
    Status Text: Go to the next record
    Menu Settings
     Enabled when: SalIsWindowEnabled( mdiWindow.pbNext )
    Menu Actions
     Call SalPostMsg( mdiWindow.pbNext, SAM_Click, 0, 0 )
   Menu Item: &Last
    Status Text: Go to the first record
    Menu Settings
     Enabled when: SalIsWindowEnabled( mdiWindow.pbLast )
    Menu Actions
     Call SalPostMsg( mdiWindow.pbLast, SAM_Click, 0, 0 )
   Menu Separator
   Menu Item: &Update
    Menu Actions
     Call SalPostMsg(pbUpdate, SAM_Click, 0, 0 )
   Menu Item: &Insert
    Keyboard Accelerator: Ins
    Menu Actions
     Call SalPostMsg(pbNew, SAM_Click, 0, 0 )
   Menu Item: &Delete
    Keyboard Accelerator: Del
    Menu Actions
     Call SalPostMsg(pbDelete, SAM_Click, 0, 0 )
   Menu Item: &Cancel
    Menu Actions
     Call SalPostMsg(pbUndo, SAM_Click, 0, 0 )
  Named Menu: menuMDIWindows
 Contents
  Data Field: dfMrMs
   Class: clsDfResetDirty
   Data
    Maximum Data Length: 5
  Data Field: dfFirstName
   Class: clsDfResetDirty
   Data
    Maximum Data Length: 20
```

```
Data Field: dfMiddleInitial
 Class: clsDfResetDirty
 Data
  Maximum Data Length: 1
Data Field: dfLastName
 Class: clsDfResetDirty
 Data
  Maximum Data Length: 20
Data Field: dfCompany
 Class: clsDfResetDirty
 Data
  Maximum Data Length: 40
Data Field: dfHomePhone
 Class: clsDfResetDirty
 Data
  Maximum Data Length: 20
Data Field: dfWorkPhone
 Class: clsDfResetDirty
 Data
  Maximum Data Length: 20
Data Field: dfAlternatePhone
 Class: clsDfResetDirty
 Data
  Maximum Data Length: 20
Data Field: dfStreet1
 Class: clsDfResetDirty
 Data
  Maximum Data Length: 30
Data Field: dfStreet2
 Class: clsDfResetDirty
 Data
  Maximum Data Length: 30
Data Field: dfStreet3
 Class: clsDfResetDirty
 Data
  Maximum Data Length: 30
Data Field: dfCity
 Class: clsDfResetDirty
 Data
  Maximum Data Length: 20
Data Field: dfState
 Class: clsDfResetDirty
Data Field: dfZip
 Class: clsDfResetDirty
 Data
```

```
   Maximum Data Length: 10
 Data Field: dfCountry
  Class: clsDfResetDirty
  Data
   Maximum Data Length: 20
 Data Field: dfBirthday
  Class: clsDfResetDirty
  Data
   Maximum Data Length: 10
 Data Field: dfAnniversary
  Class: clsDfResetDirty
  Data
   Maximum Data Length: 10
 Data Field: dfOtherOccasion
  Class: clsDfResetDirty
  Data
   Maximum Data Length: 10
 Data Field: dfOtherDay
  Class: clsDfResetDirty
  Data
   Maximum Data Length: 10
 Multiline Field: mlNotes
  Class: clsMlResetDirty
  Data
   Maximum Data Length: 1024
  Display Settings
   Word Wrap? Yes
 Pushbutton: pbUpdate
  Title: &Update
  Picture File Name: UPDATE.BMP
  Message Actions
   On SAM_Click
    Call SalPostMsg(hWndForm, PM_Update,0,0 )
 Pushbutton: pbNew
  Title: &Insert
  Picture File Name: INSERT.BMP
  Message Actions
   On SAM_Click
    Call SalPostMsg( hWndForm, PM_New,0,0 )
 Pushbutton: pbDelete
  Title: &Delete
  Picture File Name: DELETE.BMP
  Message Actions
   On SAM_Click
    Call SalPostMsg(hWndForm, PM_Delete,0,0 )
```

```
     Pushbutton: pbUndo
      Title: &Cancel
      Picture File Name: UNDO.BMP
      Message Actions
       On SAM_Click
        Call SalPostMsg(hWndForm, PM_Undo,0,0 )
     Background Text: Mr/Ms..
     Background Text: First Name
     Background Text: MI
     Background Text: Last Name
     Background Text: Company Name
     Background Text: Home Phone
     Background Text: Work Phone
     Background Text: Alternate Phone
     Background Text: Birthday
     Background Text: Anniversary
     Background Text: Other
     Background Text: Street Address
     Background Text: City
     Background Text: State
     Background Text: Country
     Background Text: ZIP
     Background Text: Notes
     Frame
     Frame
     Frame
     Frame
    Functions
     Function: OkToLoseChangesIfAny
      Description: See if the form is dirty (changes made
        to any data fields). If yes, ask user
        if it's ok to lose changes.
      Returns
       Boolean:
      Actions
       If bFormDirty and
         SalMessageBox( 'Lose Changes?', 'Confirmation',
         MB_YesNo | MB_IconQuestion | MB_DefButton2) = IDNO
        Return FALSE
       Else
        Set bFormDirty = FALSE
        Return TRUE
    Window Variables
     ! When bFormDirty is TRUE, some fields have
       changed on the form
```

```
    Boolean: bFormDirty
    ! Remember where we are in the result set
    Number: nRowNumber
    ! When bInsert is TRUE it's insert operation
      not update
    Boolean: bInsert
    : hSqlFormSelect
     Class: clsSqlHandleSelectPhAd
    : hSqlFormUpdate
     Class: clsSqlHandlePhAd
    : hSqlFormInsert
     Class: clsSqlHandlePhAd
    : hSqlFormDelete
     Class: clsSqlHandlePhAd
    String: strSelectForm
    String: strUpdateForm
    String: strInsertForm
    String: strDeleteForm
    Boolean: bOk
    String: strUserPrompt
    String: currentRowID
  Message Actions
    On SAM_Create
     ! We are not inserting a new record
     Set bInsert = FALSE
     ! The form is not dirty
     Set bFormDirty = FALSE
     ! Disable Close from the System Menu
     Call DisableClose( hWndForm )
     ! Initialize the clsSqlHandle class with the database
       name, user name, and password
     Call hSqlFormSelect.InitializeClass(SqlDatabase,
       SqlUser, SqlPassword)
     ! The SELECT statement for the form window
     Set strSelectForm =
      'SELECT LASTNAME, MRMS, FIRSTNAME, MIDDLEINITIAL,
       COMPANY, HOMEPHONE, WORKPHONE, ALTERNATEPHONE,
       STREET1, STREET2, STREET3, CITY, STATE,
       ZIP, COUNTRY, BIRTHDAY, ANNIVERSARY,
       OTHERDAY, OTHEROCCASION, NOTES, ROWID
       INTO :dfLastName, :dfMrMs, :dfFirstName, :dfMiddleInitial,
       :dfCompany, :dfHomePhone, :dfWorkPhone, :dfAlternatePhone,
       :dfStreet1, :dfStreet2, :dfStreet3, :dfCity, :dfState,
       :dfZip, :dfCountry, :dfBirthday, :dfAnniversary,
       :dfOtherDay, :dfOtherOccasion, :mlNotes, :currentRowID
       FROM PHAD ORDER BY LASTNAME ASC, FIRSTNAME ASC'
```

```
    ! Now initialize the SQL statement
    Call hSqlFormSelect.Initialize(strSelectForm)
    ! Time to connect the sql handle
    Call hSqlFormSelect.Connect()
    ! Set DBP_PRESERVE to TRUE
    Call hSqlFormSelect.SetParameter(DBP_PRESERVE, TRUE, '')
    ! Initialize the sql statement for the UPDATE operation
    Set strUpdateForm =
      'UPDATE PHAD SET LASTNAME=:dfLastName, MRMS=:dfMrMs,
      FIRSTNAME=:dfFirstName, MIDDLEINITIAL=:dfMiddleInitial,
      COMPANY=:dfCompany, HOMEPHONE=:dfHomePhone,
      WORKPHONE=:dfWorkPhone, ALTERNATEPHONE=:dfAlternatePhone,
      STREET1=:dfStreet1, STREET2=:dfStreet2,
      STREET3=:dfStreet3,
      CITY=:dfCity, STATE=:dfState, ZIP=:dfZip,
      COUNTRY=:dfCountry,
      BIRTHDAY=:dfBirthday, ANNIVERSARY=:dfAnniversary,
      OTHERDAY=:dfOtherDay, OTHEROCCASION=:dfOtherOccasion,
      NOTES=:mlNotes WHERE ROWID = :currentRowID'
    Call hSqlFormUpdate.Initialize(strUpdateForm)
    Call hSqlFormUpdate.Connect()
    ! Initialize the sql statement for the DELETE operation
    Set strDeleteForm =
      'DELETE FROM PHAD WHERE ROWID = :currentRowID'
    Call hSqlFormDelete.Initialize(strDeleteForm)
    Call hSqlFormDelete.Connect()
    ! Initialize the sql statement for the INSERT operation
    Set strInsertForm =
      'INSERT INTO PHAD (LASTNAME, MRMS, FIRSTNAME,
      MIDDLEINITIAL, COMPANY, HOMEPHONE, WORKPHONE,
      ALTERNATEPHONE, STREET1, STREET2, STREET3, CITY,
      STATE, ZIP, COUNTRY, BIRTHDAY, ANNIVERSARY, OTHERDAY,
      OTHEROCCASION, NOTES)
      VALUES (:dfLastName, :dfMrMs, :dfFirstName,
      :dfMiddleInitial,
      :dfCompany, :dfHomePhone, :dfWorkPhone, :dfAlternatePhone,
      :dfStreet1, :dfStreet2, :dfStreet3, :dfCity, :dfState,
      :dfZip, :dfCountry, :dfBirthday, :dfAnniversary,
      :dfOtherDay, :dfOtherOccasion, :mlNotes)'
    Call hSqlFormInsert.Initialize(strInsertForm)
    Call hSqlFormInsert.Connect()
  On SAM_Activate
    If wParam = TRUE
      ! The form window is being activated
      Set bFormActivated = TRUE
    Else
```

```
    ! The form window is being deactivated
    Set bFormActivated = FALSE
  On PM_Refresh
   If OkToLoseChangesIfAny(  )
    ! Display the status on the status bar of MDI
      Refreshing may take a while, so wait cursor
    Call BeginOperation( 'Refreshing Form...' )
    If not hSqlFormSelect.Prepare()
     Call SalMessageBox(
       'Could not prepare the Select statement',
       'Serious Error', MB_Ok | MB_IconStop )
     ! Normal cursor and no status text
     Call EndOperation(  )
     ! Time to quit - send the message, not post
     Call SalSendMsg( hWndForm, SAM_Close, 0, 0 )
    If hSqlFormSelect.Execute()
     ! Mark bRollback as FALSE as the SELECT was just
       executed.
     Set bRollback = FALSE
     ! Execute was successful; now fetch the first
       record
     Call SalPostMsg( hWndForm, PM_GoToFirst, 0, 0 )
    ! Normal cursor and no status text
    Call EndOperation(  )
  On PM_GoToFirst
   If OkToLoseChangesIfAny(  )
    ! Fetch the first row
    Set bOk = hSqlFormSelect.First(nReturn)
    If bOk
     ! We are not in the middle of insert
     Set bInsert = FALSE
     ! Remember where we are in the result set
     Set nRowNumber = 0
     ! Enable Next and Last. Disable First and Previous
     Call SalEnableWindow( mdiWindow.pbNext )
     Call SalEnableWindow( mdiWindow.pbLast )
     Call SalDisableWindow( mdiWindow.pbFirst )
     Call SalDisableWindow( mdiWindow.pbPrev )
  On PM_GoToPrev
   If OkToLoseChangesIfAny(  )
    ! Fetch the previous row of the result set.
    Set bOk = hSqlFormSelect.Previous(nReturn)
    If bOk
     ! We are not in the middle of insert
     Set bInsert = FALSE
```

```
    ! Remember where we are in the result set
    Set nRowNumber = nRowNumber - 1
    ! Enable Next and Last
    Call SalEnableWindow( mdiWindow.pbLast )
    Call SalEnableWindow( mdiWindow.pbNext )
   Else
    ! Could not go to the previous row (first row?).
    If nReturn = FETCH_EOF
     Call SalDisableWindow( mdiWindow.pbFirst )
     Call SalDisableWindow( mdiWindow.pbPrev )
On PM_GoToNext
 If OkToLoseChangesIfAny(  )
  ! Fetch the next row of the result set.
  Set bOk = hSqlFormSelect.Next(nReturn)
  If bOk
   ! We are not in the middle of insert
   Set bInsert = FALSE
   ! Remember where we are in the result set
   Set nRowNumber = nRowNumber + 1
   Call SalEnableWindow( mdiWindow.pbFirst )
   Call SalEnableWindow( mdiWindow.pbPrev )
  Else
   ! Could not go to the next row (last row?).
   If nReturn = FETCH_EOF
    Call SalDisableWindow( mdiWindow.pbNext )
    Call SalDisableWindow( mdiWindow.pbLast )
On PM_GoToLast
 If OkToLoseChangesIfAny(  )
  Set bOk = hSqlFormSelect.Last(nReturn)
  If bOk
   ! We are not in the middle of insert
   Set bInsert = FALSE
   ! Remember where we are in the result set (last row, 0-
     based)
   Set nRowNumber = hSqlFormSelect.GetResultSetCount()
   ! Disable Next and Last. Enable First and Previous
   Call SalEnableWindow( mdiWindow.pbFirst )
   Call SalEnableWindow( mdiWindow.pbPrev )
   Call SalDisableWindow( mdiWindow.pbNext )
   Call SalDisableWindow( mdiWindow.pbLast )
On PM_GoToLParam
 If OkToLoseChangesIfAny(  )
  ! Fetch the row# lParam
  Set bOk = hSqlFormSelect.FetchRow(lParam, nReturn)
  If bOk
```

```
        ! We are not in the middle of insert
        Set bInsert = FALSE
        ! Remember where we are in the result set
        Set nRowNumber = lParam
        If nRowNumber = 0
         Call SalDisableWindow( mdiWindow.pbFirst )
         Call SalDisableWindow( mdiWindow.pbPrev )
         Call SalEnableWindow( mdiWindow.pbLast )
         Call SalEnableWindow( mdiWindow.pbNext )
        Else If nRowNumber =
          hSqlFormSelect.GetResultSetCount()-1
         Call SalDisableWindow( mdiWindow.pbLast )
         Call SalDisableWindow( mdiWindow.pbNext )
         Call SalEnableWindow( mdiWindow.pbFirst )
         Call SalEnableWindow( mdiWindow.pbPrev )
        Else
         Call SalEnableWindow( mdiWindow.pbFirst )
         Call SalEnableWindow( mdiWindow.pbPrev )
         Call SalEnableWindow( mdiWindow.pbLast )
         Call SalEnableWindow( mdiWindow.pbNext )
   On PM_Update
    ! See if we are in the middle of insert operation
    If bInsert
     ! Insert operation
     Call BeginOperation( 'Inserting the record...' )
     If hSqlFormInsert.Prepare() and hSqlFormInsert.Execute()
      ! Time to commit the transaction.
      Call BeginOperation( 'Committing...' )
      Call hSqlFormInsert.Commit()
      ! Mark the form as not dirty (no changes yet)
      Set bFormDirty = FALSE
      ! We are not in the middle of the insert
      Set bInsert = FALSE
      ! Now re-fetch the current row of the result set.
        In case, the UPDATE failed last time and
        rollback occurred, don't fetch the row as
        the SELECT may not have survived the
        rollback.
      If not bRollback
       Call hSqlFormSelect.FetchRow(nRowNumber, nReturn)
    Else
     ! Normal Update - not insert operation
     Call BeginOperation( 'Updating the record...' )
     If not hSqlFormUpdate.Prepare()
      Call SalMessageBox(
        'Could not prepare the Update statement',
```

```
        'Serious Error', MB_Ok | MB_IconStop )
      ! Time to quit
      Call SalSendMsg( hWndForm, SAM_Close, 0, 0 )
     If hSqlFormUpdate.Execute()
      ! UPDATE was successful, now COMMIT it.
      Call BeginOperation( 'Committing...' )
      Call hSqlFormUpdate.Commit()
      ! Mark the form as not dirty (no changes yet)
      Set bFormDirty = FALSE
      ! Now re-fetch the same row of the result set.
        One reason we have been keeping track of
        nRowNumber. This refetching is necessary to get
        the new ROWID so that if the user makes a change
        to this record again without first refreshing,
        invalid ROWID error would not occur.
        In case, the UPDATE failed last time and
        rollback occurred, don't fetch the row as
        the SELECT may not have survived the
        rollback.
      If not bRollback
       Call hSqlFormSelect.FetchRow(nRowNumber, nReturn)
   Call EndOperation(  )
  On PM_New
   If OkToLoseChangesIfAny(  )
    Call BeginOperation( 'Creating a new Form...' )
    ! It will not be an ordinary update - it's insert
    Set bInsert = TRUE
    ! Send message to all the children to initialize
      themselves
    Call SalSendMsgToChildren( hWndForm, PM_Initialize, 0, 0 )
    ! Mark the form as not dirty
    Set bFormDirty = FALSE
    Call EndOperation(  )
  On PM_Delete
   ! Get the confirmation for the delete operation
   If SalMessageBox( 'Are you sure?', 'Confirmation',
     MB_YesNo | MB_IconQuestion | MB_DefButton2) = IDYES
    ! See if we are in the middle of insert operation
    If not bInsert
     ! Not in the middle of insert operation
     Call BeginOperation( 'Deleting the record...' )
     If not hSqlFormDelete.Prepare()
      Call SalMessageBox(
        'Could not prepare the Delete statement',
        'Serious Error', MB_Ok | MB_IconStop )
```

```
      ! Time to quit
      Call SalSendMsg( hWndMDI, SAM_Close, 0, 0 )
     If hSqlFormDelete.Execute()
      ! Executed the DELETE statement. Now COMMIT it.
      Call BeginOperation( 'Committing...' )
      Call hSqlFormDelete.Commit()
      ! Mark the form as not dirty (no changes yet)
      Set bFormDirty = FALSE
      ! Current record is deleted. Display the previous
        one. If the deleted record was the first one,
        display the next one. Don't do anything if the
        transaction has been rolledback.
      If not bRollback
       If nRowNumber > 0
        Call SalPostMsg(hWndMDI,PM_GoToLParam,0, nRowNumber-1 )
       Else
        Call SalPostMsg(hWndMDI,PM_GoToLParam,0, nRowNumber+1 )
     Call EndOperation(  )
    Else
     ! In the middle of insert operation
       The record is not INSERTed yet, so simply undo
       all the changes to this record.
     Call SalPostMsg( hWndForm, PM_Undo, 0,0)
On PM_Undo
 If not bFormDirty and not bInsert
  ! Not an insert and no changes to fields
  Call SalMessageBeep( 0 )
  Call SalMessageBox(
     'No changes to this record. Nothing to undo.',
     'Error', MB_Ok | MB_IconExclamation )
 Else
  ! Mark the form as not dirty (no changes yet)
  Set bFormDirty = FALSE
  ! Now re-fetch the same row of the result set.
    One of the reasons we have been keeping track
    of nRowNumber
  Call SalPostMsg( hWndMDI, PM_GoToLParam, 0, nRowNumber )
On PM_Dirty
 Set bFormDirty = TRUE
On SAM_Close
 If not OkToLoseChangesIfAny(  )
  ! Retrun FALSE only if the form is dirty and the
    user doesn't want to lose changes. Otherwise,
    it's ok to leave.
  Return FALSE
```

```
      Else
       Return TRUE
    On SAM_Destroy
      ! Time to disconnect all the sql handles
      Call hSqlFormSelect.Disconnect()
      Call hSqlFormUpdate.Disconnect()
      Call hSqlFormDelete.Disconnect()
      Call hSqlFormInsert.Disconnect()
```

Listing 5.10 Relevant portions of code for the form window.

6
Table Windows

About Table Windows

If you have run the PhAd application which was developed in Chapter 5, you know what a table window looks like. A table window displays data in row and column format. It resembles a spreadsheet. You can use a table window in several situations:

- Displaying results of a query. Since a table window clearly represents a database table (relation), it is particularly suited for displaying results of a query.
- Browsing through rows of data. Since a table window can display multiple rows and columns at a time, it is useful when a user wants to look at several records (rows) at a time. Going back to Chapter 5, a user of PhAd may find it inconvenient to browse through records one at a time had I not provided the table window. Using the table window, a user can see several persons (records) at a time and can scroll up and down to see more persons (records).
- Inserting, updating, or deleting multiple records. A user may want to make several such changes before committing the changes or discarding all the changes. In case of multiple changes, it is simply inconvenient for a user to use a form window to make changes to one record at a time.

Types of Table Windows

A table window can be a top-level window or a child of a form or dialog box.

- A top-level table window has the same features as form windows such as title, menu, icon, and accessories (toolbar, and status bar). Note that the table window I have used in Chapter 5 is a top-level table window even though it is a child of the MDI window.

- A child table window is a child of a form window or dialog box. A child table window is like a top-level window but it does not have a menu, title, system menu, icon, toolbar, or status bar. A child table window does not have minimize and maximize push buttons in the upper, right corner like a top-level table window. Most importantly, a child table window is not resizable. In some ways a child table window is like other child objects such as a data field or a multiline text field. For example, a child table window is created and destroyed along with its parent.

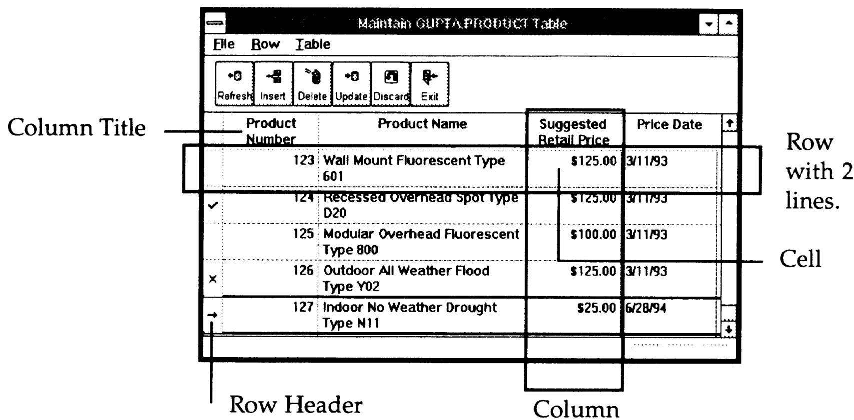

Figure 6.1 A table window and its components.

Figure 6.1 shows a top-level table window. It also shows different components of a table window such as a column, row, cell, row header, and column title. The discussion in this chapter applies to both top-level as well as child table windows.

Lines Per Row

Beginning with Release 5.0, a table window row can have multiple lines per row. By default, the number of lines per row is one. When you set it to two or more, text wraps in columns where you turn on Word Wrap (see below).

Allow Row Sizing

If you set this item to Yes in the customizer, users can drag-size rows at runtime. You can also change this setting dynamically at runtime by calling SalTblDefineRowHeader and specifying TBL_RowHdr_RowsSizable.

Word Wrap

When the lines per row is more than one, you can set this attribute for a table window column to Yes so that text in the cell wraps automatically.

Cell Type

Beginning with Release 5.0, in the customizer of a column, you can set this item to one of these values:

- *Standard.* Same behavior as SQLWindows Release 4.1 columns.
- *Drop Down List.* Behaves like a combo box when a cell gets the focus. When you set this, you can set attributes for the list.
- *Popup Edit.* Behaves like a multiline field when a cell gets the focus.
- *Check Box.* Behaves like a check box when a cell gets the focus. When you set this, you can set attributes for the check box. At runtime, a user cannot enter characters in this cell type. Instead, a user checks or unchecks it.

Table Window Cache

Understanding table window cache is very important before writing any application that uses table windows. In this section, I describe the basic concepts of a table window cache. I am using a few terms first, such as row flags, before explaining them. I hope their use is quite intuitive. Later, I explain them in detail.

Although not necessary, in most cases the rows that are displayed in a table window come from a database result set generated by some SQL query such as:

'SELECT LASTNAME, MRMS, FIRSTNAME, MIDDLEINITIAL FROM PHAD ORDER BY LASTNAME ASC, FIRSTNAME ASC'

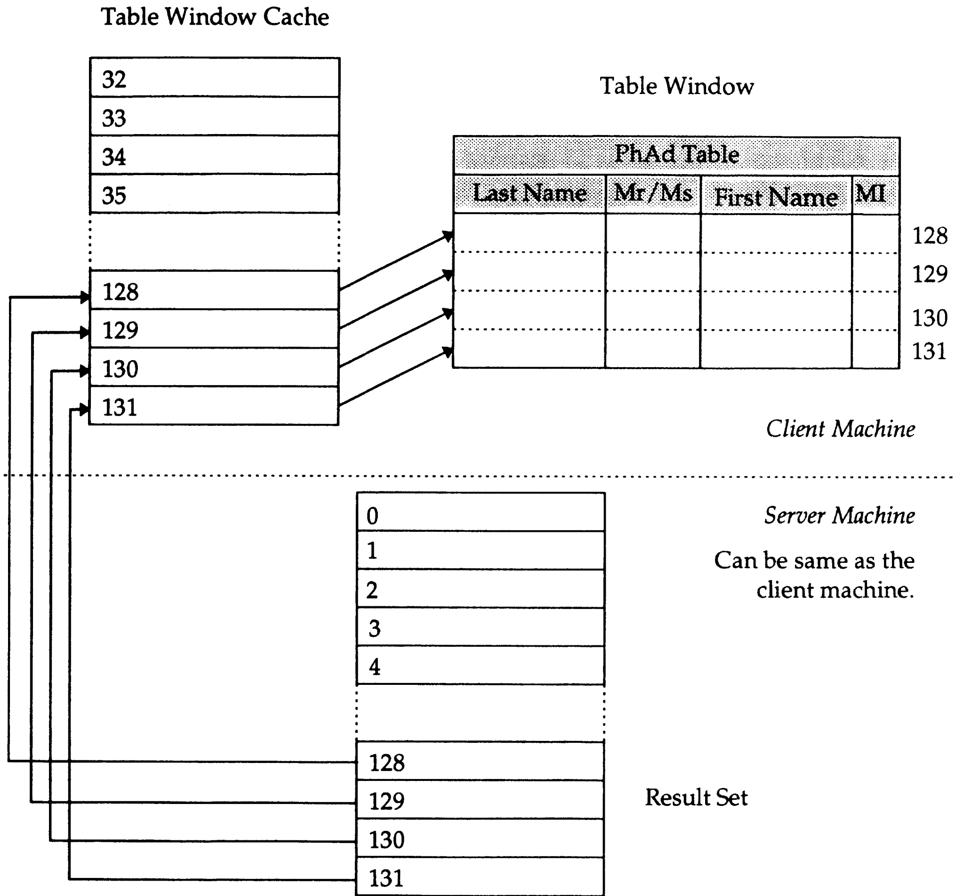

Figure 6.2 Table window cache, result set, and table window.

As seen in Figure 6.2, table window cache is the memory of the client machine used to store rows that are in the table window. There can be up to 32K rows in the table window cache *at any one time*. Even though the maximum capacity of a table window cache is 32K rows, the default value is 100 rows. This can be changed by changing the "Max Rows in Memory" item in the Customizer. System memory can also further limit the number of rows in the cache.

The cache determines how many rows can be in a table window *at any one time*. However, if the table window cache is discardable, the table window can address 32K rows. If the table window is split and the table window cache is discardable, the table window can address 32K rows in the top half and 32K rows in the bottom half. Whether or not rows are discardable depends on the "Discardable" setting for the table window in the Customizer. SQLWindows discards rows from the cache when a row needs to be fetched and the cache limit has been reached under the following conditions:

- The "Discardable" item in the Customizer is set to Yes.
- The "Max Rows in Memory" item in the Customizer defines a cache limit. This limit applies to all rows except those with a modified (ROW_Edited) flag and those with an inserted (ROW_New) flag. Modified and inserted rows are only discarded when they are deleted.

Figure 6.2 shows the state of the cache and the table window after a query that generated 132 rows in the result set and the user has scrolled to the last row in the table window by looking at each row sequentially. It is assumed that the "Max Rows in Memory" is set to 100 and the "Discardable" item is set to Yes. Notice that rows 0 through 31 of the result set are not in the cache anymore. At this point, if the user scrolls the table window up by one row, row 127 would become visible and the table window can get this row from the cache itself. However, if the user scrolls the table window to the top by dragging the scroll bar thumb to the top, rows 0 through 3 would become visible and table window would not find them in the cache. In this situation, rows 0 through 3 would be fetched from the database result set on the server.

PHAD.APP—the Table Window

Let us revisit the application I had developed in Chapter 5. PhAd uses a top-level window to display the last name, Mr/Ms, first name, and middle initial of all persons (records) in the PHAD table. Let me show you the code behind it.

Defining the Table Window

Listing 6.1 shows the relevant portions of the outline for the table window mditblTable. Notice that I have specified "Maximum Rows in Memory" to be 100 and "Discardable" to be Yes. For up to 100 rows in the PHAD table, this will cause a row to be fetched only once from the database and reduces the network traffic. Since the rows are discardable, the table window will be able to handle more than 100 rows by discarding some from the cache.

Because a row may come from the table window cache, each record is not fetched directly from the server. This means that the records in the table window may contain stale data.

I have used the named menus defined in the Global Declarations sections of PHAD.APP to define the menu for the table window. Whenever the table window is the active child, this menu takes effect and is displayed as the menu of the MDI window. See Chapter 5 for more details on these named menus.

The table window's contents include four columns: colLastName, colMrMs, colFirstName, and colMiddleInitial. I have defined these columns to be non-editable.

The function GetFocusRowStr returns the columns of the focus row concatenated in one string. I will explain the concept of a focus row and the function later in the chapter.

The table window defines some variables. Remember that the sql handle for populating the table window is defined in the Global Declarations sections of the application outline and is connected during the initial login process. See Chapter 5 for more details. Columns are like other window objects such as a data field, push button, or multi-line text field and have handles to refer to them. hWndColumn is a window handle variable which I use later to store a column's window handle.

```
Table Window: mditblTable
 Title: PhAd Table
 Display Settings
  Automatically Created at Runtime? Yes
 Memory Settings
  Maximum Rows in Memory: 100
  Discardable? Yes
 Description:
   This table window displays all the records
   of the database in a table window.
 Menu
  Named Menu: menuFile
  Named Menu: menuEdit
  Named Menu: menuMDIWindows
 Contents
  Column: colLastName
   Title: Last Name
   Editable? No
   Maximum Data Length: 20
   Data Type: String
  Column: colMrMs
   Title: Mr / Ms
   Editable? No
   Maximum Data Length: 5
   Data Type: String
  Column: colFirstName
   Title: First Name
   Editable? No
   Maximum Data Length: 20
   Data Type: String
  Column: colMiddleInitial
   Title: M I
   Editable? No
   Maximum Data Length: 1
   Data Type: String
 Functions
  Function: GetFocusRowStr
 Window Variables
  String: strSelectAllTable
  Boolean: bPopulateTbl
  Window Handle: hWndColumn
  Number: nFocusRow
  ! variables used during search by letters A..Z
  String: strLastName
  Number: nLastNameFirstChar
```

```
String: strLetterA
Number: nLetterA
! variables used for binary search
Number: nResultSetCount
Number: nRowLow
Number: nRowHigh
Boolean: bFound
Number: nContextRow
```

Listing 6.1 Definition of the table window – columns, menus, function, and variables.

Initialization—On SAM_Create

Listing 6.2 shows the code that is executed upon receiving a SAM_Create message. I process this message to do initialization such as calculating the ASCII value of 'A' to be used later, and disabling the Close menu item from the system menu. I also initialize the SQL statement which is used to fetch records from the database into the table window. The records are sorted by last name and first name in the ascending order. Since I specify LASTNAME before FIRSTNAME, the rows are sorted by the last name first; only if there are more than one rows with the same last name is the first name used to sort them further. If there are two or more rows with the same last and first names, they appear in any order depending on the order in which these records were inserted into the database.

Notice that I have not used any INTO variables to read the table fields LASTNAME, MRMS, FIRSTNAME, and MIDDLEINITIAL. Since the contents section of the table window defines columns colLastName, colMrMs, colFirstName, and colMiddleInitial exactly in the same order and of the same data type as the columns in the query, it is not necessary to specify the INTO variables. If I wanted to specify the INTO variables, I could use the following SQL statement with the same results:

```
'SELECT LASTNAME, MRMS, FIRSTNAME, MIDDLEINITIAL INTO
:colLastName, :colMrMs, :colFirstName, :colMiddleInitial
FROM PHAD ORDER BY LASTNAME ASC, FIRSTNAME ASC'
```

Notice the use of the column names just like any other variables.

In both these cases, the columns were created at designtime. As a third case, I could also use the SQL statement as shown in the listing 6.2 without actually

creating any columns at designtime. In such a case, SQLWindows itself would create the columns at runtime based on the columns in the query. Such a table window is referred to as the Dynamic Table Window. Since the columns do not have names, there are some differences in how you can reference them in SQL statements and while calling SAL functions. Also, regardless of the data type of the columns in the query, columns in a dynamic table window are always of type String.

```
On SAM_Create
  ! ASCII value of the letter A for later use
  Set strLetterA = 'A'
  Set nLetterA = SalStrLop(strLetterA)
  ! Disable Close from the System Menu
  Call DisableClose( hWndForm )
  ! The SELECT statement for the table window
  Set strSelectAllTable =
    'SELECT LASTNAME, MRMS, FIRSTNAME, MIDDLEINITIAL
    FROM PHAD
    ORDER BY LASTNAME ASC, FIRSTNAME ASC'
  ! Set the DBP_PRESERVE to true so that
    the result set is preserved after a COMMIT
  Call SqlSetParameter( hSqlTable, DBP_PRESERVE, TRUE, '' )
```

Listing 6.2 Code that handles a SAM_Create message.

Populating the Table Window—SalTblPopulate

The table window is populated with records from the database upon receiving a PM_Refresh message. The table is populated using the SalTblPopulate function.

SalTblPopulate

```
bOk = SalTblPopulate ( hWndTbl, hSql, strSelect, nMethod )
```

SalTblPopulate compiles, binds, and executes a SQL SELECT statement, then fetches the rows of the result set and populates a table window with them. SalTblPopulate also manages table window browsing.

hWndTbl is the window handle (or name) of the table window to populate.

hSql is the sql handle of the SELECT statement. As you know, table window cache only keeps a certain number of rows in the memory. This number is typically less than the number of rows in the result set. If the 'Discardable' attribute is set to Yes, and TBL_FillNormal or TBL_FillAllBackground is used

(see below), some of these rows may be discarded from the cache. Later, if these rows become visible in the table window, the cache needs to fetch them from the server. Hence it is important that you do not use this sql handle with other functions while the table window is in use.

strSelect is the SELECT statement. If the string is null, SQLWindows uses the previously prepared SELECT statement associated with hSql. This avoids re-preparing the statement each time SalTblPopulate executes.

Methods to Populate a Table Window

nMethod specifies how to populate the table window. You can specify one of these values:

Method	Description
TBL_FillNormal	The SalTblPopulate function uses this method to populate only the visible portion of a table window first, then returns control to the application. SQLWindows populates the rest of the table window as you scroll and brings new rows into view. This is the most commonly used method to populate a table window.
TBL_FillAllBackground	The SalTblPopulate function uses this method to populate the visible portion of a table window first, then returns control to the application and continues fetching rows in the background. SQLWindows fetches subsequent rows at a rate of one row every 1/4 of a second. Once all rows have been fetched, the table window receives a SAM_FetchDone message.
TBL_FillAll	The SalTblPopulate function uses this constant to populate an entire table window at once. When you use this method, you should know approximately how big your result set is and specify "Maximum Rows in Memory" to be at least of that size.

In this application, I do not process SAM_FetchRow messages but if you process and return from the SAM_FetchRow message, SalTblPopulate does not fetch that row into the table window. If you process but do not return from the SAM_FetchRow message, SalTblPopulate fetches the row into the table window.

The SAM_FetchRow message is sent to the table window *before* the row is actually fetched from the result set so in processing this message, you cannot refer to the row being fetched. However, the SAM_FetchRowDone message is sent *after* the row is fetched, enabling you to refer to the row just fetched.

```
On PM_Refresh
 Call BeginOperation( 'Refreshing Table...' )
 Set bPopulateTbl = SalTblPopulate( hWndForm, hSqlTable,
   strSelectAllTable, TBL_FillNormal )
 If not bPopulateTbl
  Call SalMessageBox( 'Could not populate the table',
    'Error', MB_Ok | MB_IconStop )
  Call SalSendMsg( hWndMDI, SAM_Close, 0, 0 )
 Call SalTblSetTableFlags( hWndForm,
   TBL_Flag_HScrollByCols | TBL_Flag_SingleSelection |
   TBL_Flag_SuppressLastColLine, TRUE )
 ! Reset the nResultSetCount variable
 Set nResultSetCount = 0
 Call EndOperation(  )
```

Listing 6.3 Populating the table window upon receiving a PM_Refresh message.

Table Window Flags

As you can see in listing 6.3, I set TBL_Flag_HScrollByCols, TBL_Flag_SingleSelection, and TBL_Flag_SuppressLastColLine to TRUE. Let me show you what these and other table window flags mean.

Table Window Flag	Description
TBL_Flag_EditLeftJustify	This flag instructs SQLWindows to left-justify cells while they are being edited. By default, cell editing behavior reflects the column definition. For example, when you edit a right-justified column, SQLWindows right-justifies the text.

Table Window Flag	Description
TBL_Flag_GrayedHeaders	This flag instructs SQLWindows to paint column headings and row headings with color and shading similar to the push buttons.
TBL_Flag_HScrollByCols	This flag instructs SQLWindows to horizontally scroll one column at a time. Otherwise, scrolling occurs one character at a time.
TBL_Flag_MovableCols	This flag lets the user drag-move columns.
TBL_Flag_SelectableCols	This flag lets the user select columns by clicking the column title.
TBL_Flag_ShowVScroll	This flag instructs SQLWindows to display the vertical scroll bar. Otherwise, it is only displayed when there are more rows than can fit in the table window display.
TBL_Flag_ShowWaitCursor	This flag instructs SQLWindows to display a wait cursor (hourglass) while it responds to a user action that causes scrolling of the table window. Examples of user action include pressing one of the Page Up, Page Down, Home, or End keys, and paging with the vertical scroll bar. The default setting for TBL_Flag_ShowWaitCursor is TRUE.
TBL_Flag_SingleSelection	This flag ensures that the user can only select one row at a time from a table window.
TBL_Flag_SizableCols	This flag lets the user drag-size columns.
TBL_Flag_SuppressLastColLine	This flag suppresses the last column separation line.
TBL_Flag_SuppressRowLines	This flag suppresses painting of the dotted lines that separate rows.

Browsing Through Rows

When a user presses First, Prev, Next, or Last push buttons, or chooses the same from the record popup menu, the table window receives a PM_GoToFirst, PM_GoToPrev, PM_GoToNext, or PM_GoToLast message. It is the responsibility of the table window to actually go to the appropriate row. I accomplish this by using SalTblSetRow function.

SalTblSetRow

```
nRowNum = SalTblSetRow ( hWnd, nRowPos )
```

This function sets the focus to the first, last, next, or previous row in a table window. hWnd is the handle (or name) of the window whose current focus row you want to change. nRowPos is the position of the focus row. Specify one of these values: TBL_SetFirstRow, TBL_SetLastRow, TBL_SetNextRow, TBL_SetPrevRow. This function returns the row number of the new focus row. I think it's time to explain what I mean by a focus row, selected row, and context row.

Focus Row

The focus frame is a double-line border that surrounds an entire row. The row that has the focus frame surrounding it is referred to as the focus row. To put the focus frame on a row, you can click anywhere on that row. You can use the up or down arrow key to move the focus frame to a different row.

Coming back to PhAd, if I had defined columns in the table window to be editable and a user types, the keyboard input would go to a column in the focus row. To be more precise, it goes to the column with an insertion point (focus cell). The insertion point is a blinking vertical line.

Selected Row

When a row is selected, it is displayed in inverse-video mode – it has a dark background and light characters. There can be more than one selected rows in a table window unless you have set the TBL_Flag_SingleSelection table flag using SalTblSetTableFlags function.

Row Flags

The inverse-video mode tells you visually whether a row is currently selected or not. If you want to find out programmatically, you can use SalTblQueryRowFlags to find out if a specific row has the ROW_Selected flag set. SQLWindows keeps track of the state of a table window's rows by setting or resetting the following flags. You can set or reset these flags programmatically from your application. Note that there can be more than one flag set at a time for a row.

Row Flag	Description
ROW_Edited	This flag indicates that a row was edited.
ROW_Hidden	This flag indicates that a row is hidden from view.
ROW_HideMarks	This flag indicates that the row's editing marks (in the row header) are hidden from view.
ROW_MarkDeleted	This flag indicates that the row is marked for deletion.
ROW_New	This flag indicates that the row has been newly inserted.
ROW_Selected	This flag indicates that the row is selected.
ROW_UnusedFlag1	This flag is available for your application to use.
ROW_UnusedFlag2	This flag is available for your application to use.

Context Row

The context row is the row whose individual column values are referenced when used as variables. The context row is usually the same as the focus row, but it can be different. Changing the context row does not affect the focus row.

The context row is the row used when you refer to a column in the following items:

- SQL statements.

- Function calls.
- SAL statements such as Set statement.

Before a SAM_FetchRow message is sent, SQLWindows automatically sets the context row so that assignments made while processing the SAM_FetchRow message reference the correct row.

You can set the context row with SalTblSetContext function. This function does not fetch a row automatically if it is not already in the table window cache. If the context row is set to a row not already in cache, a blank row appears in the table window. To avoid this, you can use SalTblFetchRow before setting the context. This is precisely what I do upon receiving PM_GoToLParam. See listing 6.4. Let me now explain what SalTblFetchRow does.

```
On PM_GoToFirst
 Call SalTblSetRow( hWndForm, TBL_SetFirstRow )
On PM_GoToPrev
 Call SalTblSetRow(hWndForm, TBL_SetPrevRow )
On PM_GoToNext
 Call SalTblSetRow(hWndForm, TBL_SetNextRow )
On PM_GoToLast
 Call SalTblSetRow(hWndForm, TBL_SetLastRow )
On SAM_Click
 ! lParam contains the row number
 Set nReturn = SalPostMsg
   ( hWndMDI, PM_GoToLParam, 0, lParam )
On SAM_RowHeaderClick
 ! This application wants SAM_RowHeaderClick to behave
   as SAM_Click. lParam contains the row number
 Call SalPostMsg( hWndForm, SAM_Click, wParam, lParam )
On PM_GoToLParam
 ! SalTblSetFocusRow does not fetch the row if the
   row does not exist in the cache, so fetch the row
   first.
 Set nReturn = SalTblFetchRow( hWndForm, lParam )
 Set bReturn = SalTblSetFocusRow( hWndForm, lParam )
```

Listing 6.4 Browsing through table window rows.

SalTblFetchRow

```
nResult = SalTblFetchRow ( hWndTbl, nRow )
```

This function sends a SAM_FetchRow message to a table window if the row you specify is not currently in the table window cache. To process the SAM_FetchRow message, fetch the row from the database or another data source. Since in this case I am not processing the SAM_FetchRow message, SQLWindows does the default processing and fetches the row from the database.

The SAM_FetchRow message sets the context row to the row you specify before the table window receives the message.

hWndTbl is the handle (or name) of the table window that owns the row being retrieved. nRow is the row number. nResult is one of these values: TBL_RowDeleted, TBL_RowFetched, TBL_NoMoreRows.

SAM_Click

When you click on a table window cell or perform a keyboard activity which simulates a mouse click, a SAM_Click message is sent to a table window and the column with the focus.

For a table window and column, hWndForm is the window handle of the table window. hWndItem is the window handle of the column. wParam is Unused and lParam contains the row number of the table window.

As you can see in listing 6.4, I use the row number stored in lParam to post a PM_GoToLParam message to the MDI window.

SAM_RowHeaderClick

When you click on a row in the row header, a SAM_RowHeaderClick message is sent to a table window and QuestWindow. This message is not sent unless SalTblDefineRowHeader has been used to define a row header.

While processing this message, both hWndForm and hWndItem contain the window handle of the table window. wParam is unused and lParam contains the row number of the row whose header was clicked.

In this application, since I wanted SAM_RowHeaderClick to behave the same way as SAM_Click, upon receiving SAM_RowHeaderClick, I simply post SAM_Click message to the table window passing on the value of lParam.

Row Validation—SAM_RowValidate

Consider the scenario where a PhAd user is looking at, for example, Mr. Tom Cruise's record and has made a change in his work phone number on the form. If the user forgets to press the Update push button on the form window, the database is not yet updated with the new work phone number. So, when the user clicks on a different row (or row header) in the table window, I wanted the table window to inquire from the form window if it's ok to lose the changes and move to a different row. If the user realizes that he or she doesn't want to move without first saving the changes, I wanted the table window to not move the focus away from the current row. I accomplish this by processing SAM_RowValidate message. This message is mainly used to verify the contents of the row.

```
On SAM_RowValidate
 ! This is one exception where the table
   window calls a function of the form window
   directly to enquire if it is ok to lose
   focus from the current row.
 If mdifrmOnePerson.OkToLoseChangesIfAny()
  Return VALIDATE_Ok
 Else
  Return VALIDATE_Cancel
```

Listing 6.5 SAM_RowValidate.

SAM_RowValidate is sent to a table window and QuestWindow before the table window's focus row changes, letting the application validate the contents of the row.

The SAM_RowValidate message is sent regardless of the current row flags or cell edited flags. It is sent only when you change the focus row, not when you remove the focus from the table window.

If the application returns VALIDATE_Cancel, the focus remains on the current row. If the application returns VALIDATE_Ok, the table window behaves normally and changes the focus row.

While processing this message, hWndForm and hWndItem are the window handles of the table window or QuestWindow. wParam is unused and lParam contains the number of the row which is about to lose the focus.

Searching among Table Window Rows

Listing 6.6 shows you how the table window responds to a PM_IndexLetter message and seaches for a row where the colLastName begins with a letter whose ASCII value is supplied in wParam. I use a binary search for better performance.

Binary Search

Essentially, binary search divides up the search range into two equal (almost equal if there are even number of rows) parts and a middle point. I then compare wParam with the first letter of colLastName. If wParam is less, it means that the desired row cannot be found in the upper half, and I continue the search in the lower half. If wParam is more, it means that the desired row cannot be found in the lower half, and I continue the search in the upper half. If wParam is the same as the first letter in the colLastName, I have found the desired row and I am done.

Note that we cannot assume that all rows of the table window are in the cache, so using SalTblSetContext would not serve the purpose. That is why I call SalTblFetchRow which makes sure that the context is set to the specified row and fetches the row from the database if it is not in the cache. Since the context is set to the specified row, I can use colLastName just like any other variable and it would refer to the cell at the intersection of colLastName column and the context row.

SalStrUpperX

```
strTarget = SalStrUpperX ( strSource )
```

SalStrUpperX function returns a string which is strSource converted to uppercase. Note that no change is made to strSource.

```
On PM_IndexLetter
 Set nContextRow = 0
 ! It returns the row number of a row which has
   the last name starting with the letter supplied in
   wParam. If none is found, last row with the letter
   less than wParam is returned. If there are more than
   one last names starting with the letter, it returns
   some row number from among them. It uses binary search
   for better performance.
```

```
Set nRowLow = 0
! Don't calculate result set count if we have
  already calculated it.
If nResultSetCount = 0
 If not SqlGetResultSetCount( hSqlTable, nResultSetCount )
  ! Could not get the result set count
  Set nResultSetCount = 0
! row numbers are 0-based
Set nRowHigh = nResultSetCount - 1
Set bFound = FALSE
While not bFound and (nRowLow <= nRowHigh)
 Set nContextRow = (nRowLow + nRowHigh)
 If SalNumberMod( nContextRow, 2 ) = 1
  ! the number was an odd number, make it even
  Set nContextRow = nContextRow - 1
 Set nContextRow = nContextRow/2
 ! Fetch the row if it is not in the window cache
 If SalTblFetchRow(hWndForm, nContextRow ) !=
   TBL_RowFetched
  Return 0
 Set strLastName = SalStrUpperX(colLastName)
 Set nLastNameFirstChar = SalStrLop(strLastName)
 If wParam = nLastNameFirstChar
  Set bFound = TRUE
 Else If (wParam < nLastNameFirstChar)
  ! The desired row is in the lower half
  Set nRowHigh = nContextRow-1
 Else
  ! The desired row is in the upper half
  Set nRowLow = nContextRow+1
If bFound
 Return nContextRow
Else
 Return nRowHigh
```

Listing 6.6 Searching for a row with colLastName starting with a specific letter.

Using Column Names as Variables – Setting Context

Function GetFocusRowStr defined by the table window concatenates the columns of the focus row in one string and returns them in a receive parameter strRowParm. This function first calls SalTblQueryFocus to find out which row currently has the focus. It then calls SalTblSetContext to set context to that row.

This is necessary to be able to refer to the columns colMrMs, colFirstName, colMiddleInitial, and colLastName for that row. Let me explain these two functions.

```
Function: GetFocusRowStr
 Description:
   This function returns the columns of the
   focus row concatenated in one string.
 Parameters
  Receive String: strRowParm
 Local variables
  Number: nFocusRow
  Boolean: bReturn
 Actions
  ! Set the context to the row which has the focus
  Set bReturn = SalTblQueryFocus
    ( hWndForm, nFocusRow, hWndColumn )
  If bReturn
   Call SalTblSetContext( hWndForm, nFocusRow )
  ! Initialize the string parameter
  Set strRowParm = ''
  ! Check if various columns are null. If not, append them
    strRowParm. The context is set to the row with the focus.
  If colMrMs != ''
   Set strRowParm = colMrMs || ' '
  If colFirstName != ''
   Set strRowParm = strRowParm || colFirstName || ' '
  If colMiddleInitial != ''
   Set strRowParm = strRowParm || colMiddleInitial || ' '
  If colLastName != ''
   Set strRowParm = strRowParm || colLastName
```

Listing 6.7 Concatenating columns – using column names as variables.

SalTblQueryFocus

`bOk = SalTblQueryFocus ( hWndTbl, nRow, hWndCol )`

This function identifies the cell in a table window with the focus. hWndTbl is the window handle (or name) of a table window. nRow is a receive parameter and after the call it contains the row number of the focus row. hWndCol is also a receive parameter and, after the call, it contains the window handle of the column with the focus.

bOk is TRUE if the function succeeds and FALSE if it fails.

SalTblSetContext

```
bOk = SalTblSetContext ( hWndTbl, nRow )
```

This function sets a table window's context row. Setting the context row does not send a SAM_FetchRow message. If the row is not currently in memory, SQLWindows creates a new row in memory and sets its cell values to null.

hWndTbl is the handle (or name) of a table window whose context row you want to set. nRow is the row number of the new context row.

bOk is TRUE if the function succeeds and FALSE if it fails.

Maintaining Records Using a Table Window

In the remainder of this chapter, I will develop a new application TABLE.APP to show you how you can use a table window to maintain records in a database table. Using TABLE.APP, a user can insert, delete, and modify records of the PRODUCT table of the GUPTA database. Figure 6.1 shows how the table window looks. The table window has its accessories enabled – both the toolbar and the status bar.

A user can edit an existing record by just going to a cell of the desired row and editing it. A user can insert a new record by pressing Insert push button or choosing Insert from the Record popup menu. A new, blank row appears at the bottom of the table. The user can enter values in the columns of this row. Finally, to delete a row, a user can select a row by clicking on its row header, and pressing the Delete push button or selecting Delete from the Record popup menu. Note that all these actions only affect the table window and its cache. No changes have been made to the database yet.

After making all the necessary changes, a user can either discard all the changes by pressing the Discard push button, or apply the changes to the database by pressing the Update push button. The Table popup menu also provides menu alternatives.

Let me explain how TABLE.APP accomplishes this.

Application Global Declarations

Listing 6.8 shows the global declarations for TABLE.APP. It defines the necessary constants for messages such as PM_Refresh. It also defines one sql handle

hSqlTable so that the login dialog box can connect it. Once the login is successful, it creates the table window. The application disconnects this sql handle upon receiving a SAM_AppExit message. It also defines a global error handler in the event of a SQL error.

```
Application Description: TABLE.APP
  Chapter 6
  Table Windows
  Power Programming with SQLWindows
  by Rajesh Lalwani.
  Copyright (c) 1994 by Gupta Corporation.
  All rights reserved.
 Global Declarations
  Constants
   User
    Number: PM_Refresh = SAM_User
    Number: PM_Insert = SAM_User+1
    Number: PM_Delete = SAM_User+2
    Number: PM_Update = SAM_User+3
    Number: PM_Discard = SAM_User+4
    Number: ERROR_TIMEOUT = -1805
    Number: ERROR_ROWID = 806
  Variables
   Sql Handle: hSqlTable
   Boolean: bConnectTable
   Sql Handle: hSqlError
   Number: nError
   Number: nPos
   Number: bRollback
   String: strMessage
  Application Actions
   On SAM_AppStartup
    ! Login Dialog Box with defaults for database,
      user, password. Pass hSqlTable as the receive
      parameter.
    Set bConnectTable = SalModalDialog( dlgLogin, hWndNULL,
      'GUPTA', 'SYSADM', 'SYSADM', hSqlTable )
    ! Quit the application if connect not successful
    If not bConnectTable
     Call SalQuit()
    ! Connect successful, create the MDI window
    Call SalCreateWindow( tblMain, hWndNULL )
   On SAM_AppExit
    ! Disconnect the sql handle of our first connect
    If bConnectTable
```

```
    Call SqlDisconnect( hSqlTable )
  On SAM_SqlError
    ...
```

Listing 6.8 TABLE.APP—relevant portions of outline for the application.

Table Window tblMain

Listing 6.9 shows you how I have defined the table window tblMain. Note that I have specified 'Automatically Created at Runtime?' to be No. Also, I have specified the maximum number of rows in memory to be 100 and the rows are discardable. This listing also shows you how I have defined the menus, contents of the toolbar (push buttons such as Insert, Delete), and the contents of the table window itself (columns). The menus and the push buttons simply post appropriate messages to the table window. The real work is done by corresponding message handlers which I will explain shortly.

Finally, notice that I have defined an additional column colROWID to store the ROWID of each row fetched from the database. I have defined this column to be invisible. If you want to maintain records of a database table in a multi-user environment, it is important to use the ROWID feature of SQLBase. See Chapter 3 for more details. The best way to remember the ROWID of each row of the table window is to read the ROWID into a table window column as I have done.

```
Table Window: tblMain
 Title: Maintain GUPTA.PRODUCT Table
 Accesories Enabled? Yes
 Display Settings
  Automatically Created at Runtime? No
 Memory Settings
  Maximum Rows in Memory: 100
  Discardable? Yes
 Menu
  Popup Menu: &File
   Status Text: Exit and About Menu Items
   Menu Item: E&xit
    Status Text: Exit from the Application
    Menu Actions
     Call SalPostMsg( hWndForm, SAM_Close, 0, 0 )
   Menu Item: &About
    Status Text: About Dialog Box
    Menu Actions
     Call SalModalDialog( dlgAbout, hWndForm )
```

```
    Popup Menu: &Row
     Status Text: Insert and Delete a Row
     Menu Item: &Insert
      Status Text: Insert a Row
      Menu Actions
       Call SalPostMsg( hWndForm, PM_Insert, 0, 0 )
     Menu Item: &Delete
      Status Text: Delete the selected row
      Menu Actions
       Call SalPostMsg( hWndForm, PM_Delete, 0, 0 )
    Popup Menu: &Table
     Status Text: Apply or Discard Changes
     Menu Item: &Refresh
      Status Text: Refresh the Table
      Menu Actions
       Call SalPostMsg( hWndForm, PM_Refresh, 0, 0 )
     Menu Item: &Apply Changes (Update)
      Status Text: Apply Changes to the Database
      Menu Actions
       Call SalPostMsg( hWndForm, PM_Update, 0, 0 )
     Menu Item: &Discard Changes
      Status Text: Discard Changes Made
      Menu Actions
       Call SalPostMsg( hWndForm, PM_Discard, 0, 0 )
   Toolbar
    Contents
     Pushbutton: pbRefresh
      Title: Refresh
      Message Actions
       On SAM_Click
        Call SalPostMsg( hWndForm, PM_Refresh, 0, 0 )
     Pushbutton: pbInsert
      Title: Insert
      Message Actions
       On SAM_Click
        Call SalPostMsg( hWndForm, PM_Insert, 0, 0 )
     Pushbutton: pbDelete
      Title: Delete
      Message Actions
       On SAM_Click
        Call SalPostMsg( hWndForm, PM_Delete, 0, 0 )
     Pushbutton: pbUpdate
      Title: Update
      Message Actions
       On SAM_Click
```

```
      Call SalPostMsg( hWndForm, PM_Update, 0, 0 )
   Pushbutton: pbDiscard
    Title: Discard
    Message Actions
     On SAM_Click
      Call SalPostMsg( hWndForm, PM_Discard, 0, 0 )
   Pushbutton: pbExit
    Title: Exit
    Message Actions
     On SAM_Click
      Call SalPostMsg( hWndForm, SAM_Close, 0, 0 )
 Contents
  Column: colProdNum
   Title: Product
     Number
   Editable? Yes
   Data Type: Number
  Column: colProdName
   Title: Product Name
   Editable? Yes
   Maximum Data Length: 50
   Data Type: String
   Justify: Left
  Column: colSRP
   Title: Suggested
     Retail Price
   Editable? Yes
   Data Type: Number
   Justify: Right
  Column: colPriceDate
   Title: Price Date
   Editable? Yes
   Data Type: Date/Time
   Justify: Left
  Column: colROWID
   Title: ROWID
   Visible? No
   Editable? No
   Data Type: String
 Window Variables
  ! Variables for inserting a record
  Sql Handle: hSqlInsert
  String: strInsert
  Boolean: bConnectInsert
  ! Variables for deleting a record
```

```
Sql Handle: hSqlDelete
String: strDelete
Boolean: bConnectDelete
! Variables for updating a record
Sql Handle: hSqlUpdate
String: strUpdate
Boolean: bConnectUpdate
! Variable for selecting records
String: strSelect
! Miscellaneous
Number: nRow
```

Listing 6.9 Table Window and toolbar, and their contents.

How to Check If There are Modified Rows

As I will show you shortly, there are times when you want to know if there are any rows that are either new, edited, or marked for deletion. And if there are any such rows, is it OK with the user to lose these changes? Listing 6.10 shows a function that I have defined for this purpose.

Function OkToLoseChangesIfAny finds out if there are any rows which are new, edited, or marked for deletion by calling SalTblAnyRows function.

SalTblAnyRows

`bAny = SalTblAnyRows ( hWndTbl, nFlagsOn, nFlagsOff )`

This function determines whether any rows in the specified table window match certain flags.

hWndTbl is the window handle (or name) of a table window. nFlagsOn are the flags that the row should have. You can combine ROW_* flags using the OR (|) operator. nFlagsOff are the flags that the row should not have. You can combine ROW_* flags using the OR (|) operator. The function returns TRUE if any of the table window's rows have *any* of the nFlagsOn flags and *none* of the nFlagsOff flags.

If you set nFlagsOn to zero (0) and nFlagsOff to zero (0), SalTblAnyRows returns TRUE if the table window contains any rows at all, regardless of their flags.

```
Functions
 Function: OkToLoseChangesIfAny
  Description:
```

```
    This function checks if there are rows
    marked for delete, insert, or have been
    edited. If none, or if the user does not
    mind losing changes, it returns TRUE.
  Returns
   Boolean:
  Actions
   ! See if there are any rows edited, new, or marked for
     deletion.
   If SalTblAnyRows( tblMain,
     ROW_Edited | ROW_MarkDeleted | ROW_New, 0 )
    Call SalMessageBeep( 0 )
    If SalMessageBox( 'Discard Changes?', 'Please Confirm',
      MB_IconQuestion | MB_YesNo | MB_DefButton2 ) = IDNO
     Return FALSE
   Return TRUE
```

Listing 6.10 Function OkToLoseChangesIfAny.

Populating the Table Window

Upon receiving a SAM_Create message, the table window initializes SQL statements for the SELECT, INSERT, DELETE, and UPDATE operations and makes the necessary database connections. Note the use of ROWID in the SQL statements. Finally, it posts a PM_Refresh message to itself.

As with PHAD.APP, I populate the table window using the SalTblPopulate function. Upon receiving a PM_Refresh message, SalTblPopulate is called. Notice that while processing the PM_Refresh message, if wParam is TRUE, it indicates that the table is to be populated again regardless of whether any modifications have been made to rows of the table window. For example, I use this feature when a user wants to discard all changes to the table window.

```
Message Actions
 On SAM_Create
  ! Select - the sql handle was connected by
    On SAM_AppStartup
  Set strSelect =
    'SELECT PROD_NUM, PROD_NAME, SRP, PRICE_DATE, ROWID
    INTO :colProdNum, :colProdName, :colSRP, :colPriceDate,
    :colROWID FROM PRODUCT'
  ! Set DBP_PRESERVE to TRUE so that the result set
    is not destroyed upon a COMMIT.
  Call SqlSetParameter( hSqlTable, DBP_PRESERVE, TRUE, '' )
```

```
    ! Use RL isolation level
    Call SqlSetIsolationLevel( hSqlTable, 'RL' )
    ! Insert
    Set bConnectInsert = SqlConnect( hSqlInsert )
    If not bConnectInsert
     Call SalQuit(  )
    Set strInsert =
      'INSERT INTO PRODUCT
      (PROD_NUM, PROD_NAME, SRP, PRICE_DATE)
      VALUES (:colProdNum, :colProdName, :colSRP, :colPriceDate)'
    ! Update
    Set bConnectUpdate = SqlConnect( hSqlUpdate )
    If not bConnectUpdate
     Call SalQuit(  )
    Set strUpdate =
      'UPDATE PRODUCT
      SET PROD_NUM=:colProdNum, PROD_NAME=:colProdName,
      SRP=:colSRP, PRICE_DATE=:colPriceDate
      WHERE ROWID=:colROWID'
    ! Delete
    Set bConnectDelete = SqlConnect( hSqlDelete )
    If not bConnectDelete
     Call SalQuit(  )
    Set strDelete =
      'DELETE FROM PRODUCT WHERE ROWID=:colROWID'
    ! Now post PM_Refresh to the the table window
    Call SalPostMsg( hWndForm, PM_Refresh, FALSE, 0 )
  On PM_Refresh
    ! If wParam is true, it is unconditional
      refresh of the table - regardless of
      whether any changes have been made.
    If wParam = TRUE or OkToLoseChangesIfAny(  )
     Call SalTblPopulate
       ( hWndForm, hSqlTable, strSelect, TBL_FillNormal )
```

Listing 6.11 Populating the table window.

Marking a Row for Deletion

First I inquire whether there are any rows that have been selected. If there are none, I simply generate a beep and tell the user that there are no rows selected. A user can select a row by clicking on the row header.

However, if there is any row which is currently selected by the user, I call SalTblSetFlagsAnyRows to set the ROW_MarkDeleted flag to be TRUE for all the

rows for which ROW_Selected flag is TRUE. This is all that is done upon receiving a PM_Delete message. A row is *marked* for deletion – the actual deletion, from the table window as well as the database, takes place when a user presses Update push button.

SalTblSetFlagsAnyRows

```
bAny =       SalTblSetFlagsAnyRows ( hWndTbl, nFlags, bSet,
             nFlagsOn, FlagsOff )
```

This function sets or clears row flags.

hWndTbl is the window handle (or name) of a table window whose row flags you want to set or clear.

nFlags contains the flags to change. You can combine flag values using the OR (|) operator.

bSet specifies whether to set (TRUE) or clear (FALSE) the specified flags.

nFlagsOn are the flags that the rows should have. You can combine flag values using the OR (|) operator. Set nFlagsOn to zero (0) and nFlagsOff to zero (0) to specify all rows.

nFlagsOff are the flags that the rows should not have. You can combine flag values using the OR (|) operator. Set nFlagsOn to zero (0) and nFlagsOff to zero (0) to specify all rows.

bAny is TRUE if any of the table window's rows have any of the nFlagsOn flags and none of the nFlagsOff flags.

```
On PM_Delete
 If SalTblAnyRows( hWndForm, ROW_Selected, 0 )
  ! Mark all the selected rows for deletion.
  Call SalTblSetFlagsAnyRows
    ( hWndForm, ROW_MarkDeleted, TRUE, ROW_Selected, 0 )
 Else
  Call SalMessageBeep( 0 )
  Call SalMessageBox ('No rows are selected.', 'Error',
       MB_IconExclamation | MB_Ok)
```

Listing 6.12 Marking the selected row for deletion.

Inserting a New Row

First, I call SalTblInsertRow to create a new row at the end of the table window.

SalTblInsertRow

```
nNewRow = SalTblInsertRow ( hWndTbl, nRow )
```

This function inserts a new blank row into a table window.

hWndTbl is the window handle (or name) of a table window.

nRow is the row number of the new row. If this value is a valid row number within the table window range, then SQLWindows inserts a blank row at that location. If you specify TBL_MaxRow, SQLWindows appends the row to the end of the table window. If the table window is split and you specify TBL_MinSplitRow, SQLWindows appends the row to the top of the lower half of the table window.

nNewRow is the number of the new row if the function succeeds. It is equal to TBL_Error if an error occurs.

Coming back to TABLE.APP, if the SalTblInsertRow function returns TBL_Error, I generate an error message. Otherwise, the return value of the function contains the row number of the row just created. I then call SalTblSetFocusCell to set the focus (and a blinking cursor) to the colProdNum column of this new row.

SalTblSetFocusCell

```
bOk =       SalTblSetFocusCell ( hWndTbl, nRow, hWndCol,
            nEditMin, nEditMax )
```

This function sets the focus to the specified table window cell (row and column). SQLWindows puts the table window into edit mode and lets the user select a portion of the data in the cell.

hWndTbl is the window handle (or name) of a table window.

nRow is the row that receives the edit focus.

hWndCol is the window handle (or name) of the column that receives the edit focus.

nEditMin is the position of the left-most character. When used with nEditMax, this parameter lets the user select (highlight) a portion of the cell text. nEditMin

must be less than or equal to nEditMax. To select all the characters in a cell, specify zero (0) for this parameter and -1 for nEditMax.

nEditMax is the position of the right-most character. When used with nEditMin, this parameter lets the user select (highlight) a portion of the cell text. nEditMax must be greater than or equal to nEditMin. To select all the characters in a cell, specify -1 for this parameter and zero (0) for nEditMin.

bOk is TRUE if the function succeeds and FALSE if it fails.

```
On PM_Insert
 Set nRow = SalTblInsertRow( hWndForm, TBL_MaxRow )
 If nRow = TBL_Error
  Call SalMessageBeep( 0 )
  Call SalMessageBox(
    'Sorry, could not insert a new row.', 'Error',
    MB_IconExclamation | MB_Ok )
 Else
  Call SalTblSetFocusCell( hWndForm, nRow, colProdNum, 0, -1 )
```

Listing 6.13 Inserting a new row.

Applying Changes—Updating the Database

This is the place where all the real work is done. The rows that are marked for deletion are deleted from the table window as well as the database. New rows are inserted into the database. Finally, for all the rows that have been edited, the database is updated. If all goes well, the transaction is committed. If any of these steps fail, the entire transaction is rolled back, so that *no* changes are made to the database or *all* the changes are made to the database.

In this application, it is the responsibility of the user to make sure that no two rows have the same PROD_NUM. In practice, however, the application must make sure that while inserting a new row or updating an existing row, unique column contraint is not violated. One way to do this is to make such a column read-only and generate the next sequential number when a new row is inserted. I leave that as an excercise for you.

Order of DELETE, INSERT, and UPDATE Operations

Note that the order in which these three operations are performed is important.

- Delete *must* happen before the edit operation. Imagine that a user edits a row and later marks this row for deletion. This row would have both ROW_Edited, and ROW_MarkDeleted flags set. If the edit operation takes place first, the ROWID of this row would change to a new value. Now, the DELETE statement would not be able to find a row with the old ROWID still in colROWID and hence would not delete this row from the database. On the other hand, performing the delete operation first removes the row from the database as well as the table window, so during the edit operation, this row does not even exist in the table window so no edit operation is performed – it is unnecessary anyway.

  Also, if a row with PROD_NUM=101 is being deleted and another row is being edited so that it has PROD_NUM=101, performing the delete operation first would not cause the duplicate primary key error since the first row would be deleted by the time edit operation is performed. On the other hand, performing the edit operation first would result in an error because although the first row with PROD_NUM is marked for deletion in the table window, it still exists in the database.

- Delete *must* happen before the insert operation. Same reasoning as above. Performing a delete operation before the insert operation takes appropriate actions in case a new row has been inserted and later marked for deletion by the user. Performing the delete operation would not find any row in the database with the blank ROWID but it will remove the row from the table window so that the insert operation would never find this row in the table window thus avoiding the unnecessary insert operation.

  In case of a row with PROD_NUM=101 being deleted and a new row with PROD_NUM=101 being inserted, same reasoning as above dictates that the delete operation must be performed before the insert operation.

- The insert operation *should* be performed before the edit operation because a new row usually has both ROW_New and ROW_Edited flags set. Because the new row has not been added to the database yet, the row would not be found for the editing operation if it were attempted first.

  I choose the order in which the insert operation is performed before the edit operation. Also, after the insert operation, I reset the ROW_Edited flag for all the rows which has the ROW_New flag set. This way, the unnecessary edit

operation is not performed for the new rows since they were inserted into the database with the new values anyway.

SalTblDoDeletes

```
bOk = SalTblDoDeletes ( hWndTbl, hSql, nFlagsOn )
```

This function applies a SQL DELETE statement to all table window rows that have nFlagsOn flags set. You must prepare the SQL DELETE statement before calling this function. As each row is deleted from the database, SQLWindows deletes it from the table window display.

This function does not perform a COMMIT. You must perform a COMMIT to ensure that the deletions are not lost in the case of a rollback.

hWndTbl is the window handle (or name) of a table window.

hSql is the sql handle of a DELETE statement.

nFlagsOn specifies the row flags. SQLWindows uses these row flags to determine which rows to delete. You can specify either ROW_MarkDeleted or ROW_Selected.

bOk is TRUE if any rows are deleted and FALSE otherwise. bOk is also FALSE if hWndTbl or hSql is invalid.

SalTblDoInserts

```
bOk = SalTblDoInserts ( hWndTbl, hSql, bClearFlags )
```

This function applies a SQL INSERT statement to all the rows in a table window that have the ROW_New flag set. You must prepare the SQL INSERT statement before calling this function.

This function does not perform a COMMIT. You must perform a COMMIT to ensure that insertions are not lost in the case of a rollback.

hWndTbl is the window handle (or name) of a table window.

hSql is the sql handle of an INSERT statement.

bClearFlags specifies whether to reset the ROW_New flag. If TRUE, SQLWindows clears the ROW_New flag of each inserted row; if FALSE, SQLWindows does not clear the ROW_New flag of each inserted row. This is

useful for error handling; if an error occurs, you can rollback the transaction and try again.

bOk is TRUE if the function succeeds and FALSE otherwise. bOk is also FALSE if hWndTbl or hSql is invalid.

SalTblDoUpdates

```
bOk = SalTblDoUpdates ( hWndTbl, hSql, bClearFlags )
```

This function applies a SQL UPDATE statement to all table window rows with the ROW_Edited flag set. You must prepare the SQL UPDATE statement before calling this function.

This function does not perform a COMMIT. You must perform a COMMIT to ensure that the updates are not lost in the case of a rollback.

hWndTbl is the window handle (or name) of a table window.

hSql is the sql handle of an UPDATE statement.

bClearFlags specifies whether to reset the ROW_Edited flag. If TRUE, SQLWindows clears the ROW_Edited flag of each changed row; if FALSE, SQLWindows does not clear the ROW_Edited flag of each changed row. This is useful for error handling; if an error occurs, you can roll back the transaction and try again.

bOk is TRUE if the function succeeds and FALSE otherwise. bOk is also FALSE if hWndTbl or hSql is invalid.

```
On PM_Update
 If SalTblAnyRows( tblMain,
   ROW_Edited | ROW_MarkDeleted | ROW_New, 0 )
  Call SalStatusSetText( hWndForm, 'Doing DELETEs.' )
  If not SqlPrepare( hSqlDelete, strDelete ) or
   not SalTblDoDeletes( hWndForm, hSqlDelete, ROW_MarkDeleted )
   Return FALSE
  ! Time to do INSERTs
  Call SalStatusSetText( hWndForm, 'Doing INSERTs.' )
  If not SqlPrepare( hSqlInsert, strInsert ) or
    not SalTblDoInserts( hWndForm, hSqlInsert, FALSE )
   Return FALSE
  ! Insert went fine. Now reset the ROW_Edited flag for
    these new rows.
  Call SalTblSetFlagsAnyRows
```

```
    ( hWndForm, ROW_Edited, FALSE, ROW_New, 0 )
  ! Time to do updates.
  Call SalStatusSetText( hWndForm, 'Doing UPDATEs.' )
  If not SqlPrepare( hSqlUpdate, strUpdate ) or
    not SalTblDoUpdates( hWndForm, hSqlUpdate, FALSE )
   Return FALSE
  ! Update went fine. Now reset all the ROW_ flags
  Call SalTblSetFlagsAnyRows
    ( hWndForm, ROW_Edited | ROW_New, FALSE,
      ROW_Edited | ROW_New, 0 )
  ! Everything went fine. Now commit the transaction.
  Call SalStatusSetText( hWndForm, 'Committing...' )
  Call SqlCommit( hSqlTable )
  Call SalStatusSetText( hWndForm, 'Ready.' )
  ! Refresh the table window forcibly
  Call SalPostMsg (hWndForm, PM_Refresh, TRUE, 0)
```

Listing 6.14 Applying changes—updating the database

Discarding Changes

If there are any rows that have been modified by a user, I confirm with the user if he or she is sure. Using MB_YesNo creates a message box with two push buttons – Yes, and No. Use of MB_DefButton2 specifies the second push button (No) to be the default push button. If the user is sure, a PM_Refresh message is posted to the table window. Note the use of wParam set to TRUE; this forces PM_Refresh handler to populate the table window again even if there are rows which have been modified.

Again, note that I do not actually go through each row of the table window and reset the flags, and undo the changes made by the user. That is impossible as no one kept track of those changes. The table window is simply populated again.

```
On PM_Discard
 If SalTblAnyRows
    (tblMain, ROW_Edited | ROW_MarkDeleted | ROW_New, 0 )
  Call SalMessageBeep( 0 )
  If SalMessageBox( 'Are you sure?', 'Please Confirm',
    MB_IconQuestion | MB_YesNo | MB_DefButton2 ) = IDYES
   ! Force refreshing of the table window. wParam = TRUE
   Call SalPostMsg( hWndForm, PM_Refresh, TRUE, 0 )
```

Listing 6.15 Discarding all changes and populating the table again.

Disconnecting from the Database

When a user chooses Exit from the File popup menu, presses the Exit push button, or chooses Close from the system menu, a SAM_Close message is sent to the table window. If the table window returns FALSE from here, the table window is not closed. In all other cases, the table window is closed and a SAM_Destroy message is sent so that the table window can do the cleanup it wants to do. In this case, upon receiving SAM_Destroy message, I disconnect all the sql handles.

```
On SAM_Close
  If not OkToLoseChangesIfAny(  )
   Return FALSE
On SAM_Destroy
  ! Disconnect all sql handles. Note that hSqlTable
    for select will be disconnected by On SAM_AppExit.
  If bConnectInsert
   Call SqlDisconnect( hSqlInsert )
  If bConnectUpdate
   Call SqlDisconnect( hSqlUpdate )
  If bConnectDelete
   Call SqlDisconnect( hSqlDelete )
```

Listing 6.16 Disconnecting from the Database.

7
Generating Reports

In this chapter I explain how you can generate reports (report templates) and either display them on the screen or print them from a SQLWindows application. Once again, I explain the concepts by giving examples:

- For an order in ORDER_MASTER table, I generate an invoice; listing each product, quantity, suggested retail price, discount, and the invoice total. I make use of ORDER_MASTER, ORDER_DETAIL, and PRODUCT tables of the GUPTA database.
- Generation of a cross-tabular report showing the sales activity of each sales person. For each sales person, the report displays the sales activity for each quarter.
- Generation of 1"x2-5/8" mailing labels used in PHAD.APP of Chapter 5.
- Generation of multiple reports in a batch mode.

A *report template* is a .QRP file which contains the design for the final *report* when seen on the screen or printed on a printer. For brevity, however, I sometimes use the word report for a report template. If there is a chance that it may cause confusion, I use the complete word report template.

The Process of Generating a Report

This is a two step process:

- Creating a report template—a .QRP file. This can be done either by using the ReportWindows tool of SQLWindows or the Report activity of Quest. A report template designed using Quest can be used by SQLWindows. Similary, a report template designed by using ReportWindows can be used by Quest.

If you are not particular about what the report looks like, you can skip this step and generate a default report template programmatically using the SAL functions SalReportCreate and SalReportTableCreate. A more typical use of these functions is to produce a new report template which defines the

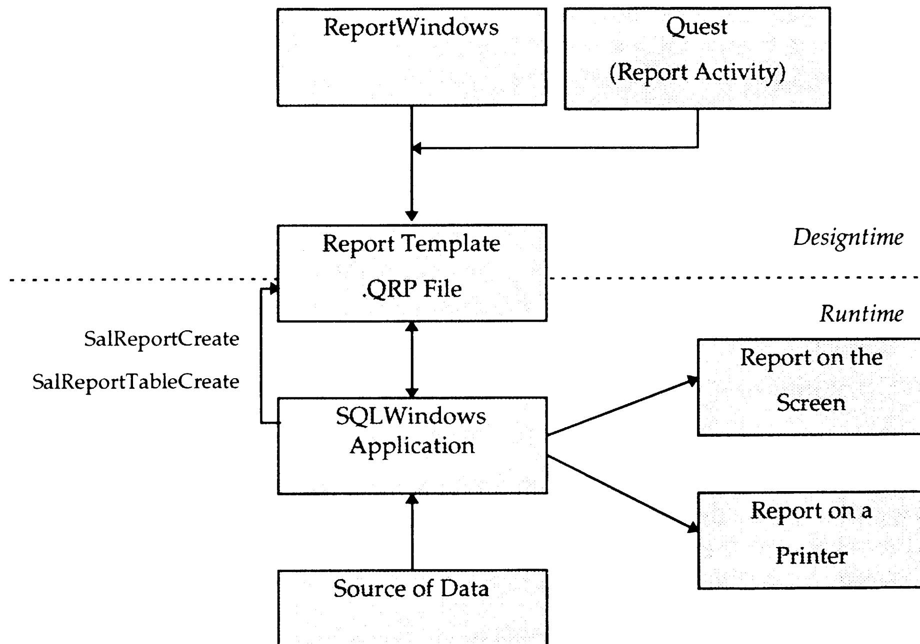

Figure 7.1 The process of generating a report.

necessary input items. Later, developers use this template and design it further using ReportWindows.

- Printing or displaying a report from a SQLWindows application. This step basically consists of launching a report and processing report messages to

feed the data to the report as needed. In most cases, the data comes from a database. If the data comes from a table window, you can avoid processing report messages by using the SAL functions SalReportTablePrint and SalReportTableView.

Figure 7.1 shows the process graphically.

Let us design our first report template; INVOICE.QRP. I use this report template in two applications – INVOICE.APP and BATCH.APP.

MECHANICAL PARTS, INC.

INVOICE

Customer: Acme Bakery
Order Num: 100
Order Date: Jan 16, 1990
Representative: 100
Terms: Net 30

	Part Number	Description	Quantity	SRP	Discount	Amount
1	100	Overhead Flood Type AA	1	$100.00	5%	$95.00
2	101	Modular Overhead Fluorescent Type 802	52	$150.00	10%	$7,020.00
3	102	Recessed Overhead Spot Type E30	173	$150.00	15%	$22,057.50

Pay this amount: $29,172.50

Figure 7.2 An invoice generated by INVOICE.APP.

Designing a Report Template for an Invoice

I design a report template INVOICE.QRP to print an invoice for an order from the ORDER_MASTER table of the GUPTA database. Each invoice begins on a fresh page. When printed, an invoice looks like the one shown in Figure 7.2. The following section provides step by step instructions on how to design this report template.

Creating a New Template

Start ReportWindows by choosing Tools, ReportWindows... from the SQLWindows menu bar. This opens ReportWindows with a new report template. Save it as INVOICE.QRP in your personal directory by choosing File, Save from the menu bar of ReportWindows. From now on, you work in ReportWindows. If you wish, you can minimize all your SQLWindows windows. If you want to take a break after any step, just save the current report. When you come back again, start SQLWindows, start ReportWindows and open INVOICE.QRP by choosing File, Open... from the menu bar.

Initially, you see a new, empty report template (see Figure 7.3). It has five blocks: Report Header, Page Header, Detail Block, Page Footer, and Report Footer.

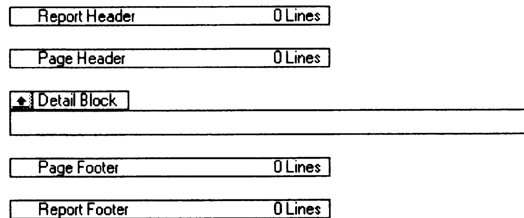

Figure 7.3 A new report template.

Initially, they are all empty except for the Detail Block which has one blank line.

A Report Header appears only in the beginning of the report. A Report Header is used to print the title of the report, if any and appears on the first page. Similarly, a Report Footer appears only at the end of the report. It usually contains report totals and summary information.

A Page Header usually contains the page number. Page Headers appear once at the top of every page except the first page. If no report header is defined, or you have specifically asked for it, the page header also appears on the first page.

Similarly, a Page Footer appears at the bottom of every page and usually contains the page number, page totals, etc.

Finally, a Detail Block prints information from individual rows showing the data in the order in which data is fed to the report at runtime.

Defining Input Items

Now you need to define Input Items for the report template. Input items are individual data items in a report. All the input items together make up an input row in the report template. They are like columns of the database table. When a SQLWindows application feeds data to a report at the run-time, the data is moved from the source of the data to the input items of the report template. These input items can be placed directly in a field in any of the template blocks or can be used in a formula, an input total variable, or a cross-tab as I show you later in this chapter. Each input item has a name and a data type.

You can define input items for a report template by choosing Format, Input, Input Items... from the menu bar. Using the Format Input Items dialog box (see Figure 7.4), define the following input items: ORDER_NUM (Number), ORDER_DATE (Date/Time), TERMS (String), REP_NUM (String), CUSTOMER (String), PROD_NUM (Number), QUANTITY (Number), DISCOUNT (Number), PROD_NAME (String), and SRP (Number).

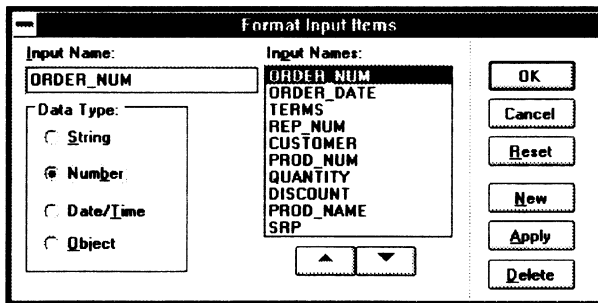

Figure 7.4 Dialog box to define or modify input items.

Defining a Break Group

When INVOICE.APP feeds data to the report at the run-time, each row contains values for all the input items you just defined: ORDER_NUM, ORDER_DATE, TERMS, REP_NUM, CUSTOMER, PROD_NUM, QUANTITY, DISCOUNT, PROD_NAME, and SRP. As you see later, the SELECT statement which fetches these rows from the database, sorts them by ORDER_NUM so that all the line

items (rows from ORDER_DETAIL) are fetched consecutively. Of course, it makes sense to print information such as ORDER_NUM, ORDER_DATE, TERMS, REP_NUM, and CUSTOMER only once for each order. The rest of the information such as PROD_NUM, QUANTITY, DISCOUNT, PROD_NAME, and SRP should be printed for each product ordered by this ORDER_NUM. To achieve this you need to define a break group for ORDER_NUM.

Break groups organize information into logical subunits. In this example, when you define a break group for ORDER_NUM, at runtime, SQLWindows carefully examines each row fed to the report for any change in ORDER_NUM. As long as the rows contain the same value of ORDER_NUM, all these rows are considered part of the same group. As soon as the value of ORDER_NUM changes from the previous row, a new group begins. This is why it is important to sort the data by the column that corresponds to the input item for the break group. You can nest break groups to eight levels.

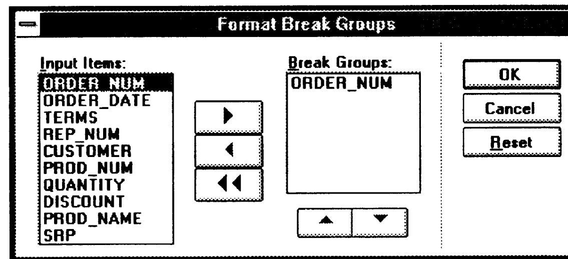

Figure 7.5 Dialog box to define break groups.

You can define a break group for ORDER_NUM by choosing Format, Break Groups... from the menu. Using the Format Break Groups dialog box (See Figure 7.5), you can move ORDER_NUM from Input Items list box to the Break Groups. In this case you only have one break group, so you don't need to use the up and down arrow push buttons at the bottom of the Break Groups list box. If you define more than one break group, the first input item in the Break Groups list box is the outermost level and the last input item the innermost. You can change position of any input item by selecting it and using these up and down arrow push buttons.

As soon as you define a break group for ORDER_NUM, you can see that two new blocks appear in the report template. One for Header:ORDER_NUM and

another for Footer:ORDER_NUM. As you see later, the header of this break group is used for printing CUSTOMER, ORDER_NUM, ORDER_DATE, REP_NUM, and TERMS while the footer of this page group is used for printing the total amount for this order. This total is calculated using an input total variable called TotalAmount.

Input Totals

In this report template, I use two input totals. TotalAmount to calculate the total amount for each order and ItemNum to calculate the item number (1, 2, 3..) for each product ordered in an order. See Figure 7.2.

Input totals, also known as two-pass totals, are totals (sum) or other aggregate types of calculations (count, minimum, maximum, average, etc.) performed by ReportWindows. ReportWindows provides the result as input totals to your report. All input totals are numeric variables which can be placed in a field or used by another formula.

Defining ItemNum Input Total

ItemNum is used to print the item number for each product ordered for a specific order. You can define this input total by choosing Format, Input, Totals... and choosing New from the Format Totals dialog box. This will bring up a Define Total dialog box as shown in Figure 7.6.

Name

I have chosen the name ItemNum for this input total.

Formula

This is the place where you choose the basic unit on which the Statistic is performed. You can simply choose an input item, or press the Formula push button and invoke the formula editor to define a formula involving input items or even other formulas. I show you how to define a formula when I discuss the next input total TotalAmount. For ItemNum I simply want to count how many products are printed up to the current row for this order, so I choose the input item PROD_NUM.

Figure 7.6 Defining an input total—ItemNum.

Statistic

Here you specify what you want ReportWindows to do with the formula specified above. You can specify the following:

Statistic
Average
Count
CountNull
CountUnique
Maximum
Minimum
Sum
Value

Since I simply want to count the number of PROD_NUM for this ORDER_NUM, I have specified Count as the statistic to perform.

Restart Event

This is the place where you can specify at what time ReportWindows should reset the input total and start calculating it afresh. You can specify First Fetch, Page Break or any break group that you defined in your report template. When

you specify First Fetch, the input total is reset when the first row is being processed by the report; it is never reset again for the rest of the rows. When you specify Page Break, the input total is reset each time a page break occurs in the report you print or display on the screen. Page Break can be used for situations such as calculating the total you forward to the next page. In addition, if you have specified any break groups in your report template, you can specify the input total to be reset when a new group begins (i.e., when the value of this input item for the current row is different from the last row processed by the report).

As you can see in Figure 7.6, I specify the Restart Event as ORDER_NUM. This is because I wanted to start with 1 for the first PROD_NUM on each order (ORDER_NUM). If I specify First Fetch instead, the numbers would grow sequentially—if the first order had 3 items, the second order would have numbers starting with 4.

Pre-Process

If the situation warrants, you can check this check box. If this check box is checked, ReportWindows calculates the input total before it formats your report. For example, if you want to calculate the total amount and print it before the line items, you must check this check box. As you can see from Figure 7.2, I have chosen to display the total amount after all the line items for the order (Footer:ORDER_NUM). It is not necessary to check this check box because by the time the footer is formatted all the rows for line items are processed and TotalAmount contains the correct total. In case of the total amount, checking the Pre-Process check box does not alter the result. It, however, affects the performance negatively because checking Pre-Process requires an additional pass of fetching data from the server.

In case of ItemNum, since I only want to print the current item number, it is important that I do *not* check the Pre-Process check box. If I check it, all the line items would have a number 3 for ItemNum if the order has 3 items.

Defining TotalAmount Input Total

This input total calculates the total amount for the entire order. For doing this it takes the help of a formula called Amount() as you can see in Figure 7.7. The formula Amount() calculates the price for each product ordered and the input total TotalAmount keeps a running a total of this for an order. Since I print this

input total in Footer:ORDER_NUM, by the time this footer is printed, the input total contains the total for the entire order.

Figure 7.7 Defining an input total—TotalAmount.

Notice that for this case I have chosen the Statistic Sum. The Restart Event is ORDER_NUM and I have not checked the Pre-Process check box.

Defining a Formula

When you are defining an input total such as TotalAmount and want to perform the Statistic on a formula instead of an input item, you can define a new formula by pressing the Formula push button. This brings up the formula editor. Figure 7.8 shows the formula editor after I have completely defined the formula Amount().

Data Items

When the formula editor comes up, the formula as seen in the top multi-line field is empty and has a generic name such as Formula1 as seen in the Formula Name data field near the bottom of this dialog box. To define a formula, you can choose the input items of the report template from the Data Items list box. This list box also contains any input totals that you have defined earlier. Double-clicking on any data item places it in the formula being defined.

Functions

In this formula, I do not use any functions. However, you can use any of the functions listed in the Functions list box. Double-clicking on a function places it in the formula along with any parameter that this function might have.

Operators

You can either type the operators *, +, -, /, | | (string concatenation) yourself or you can use the push buttons in the Operators group box.

In general, if you feel more productive by typing data items, functions, operators,

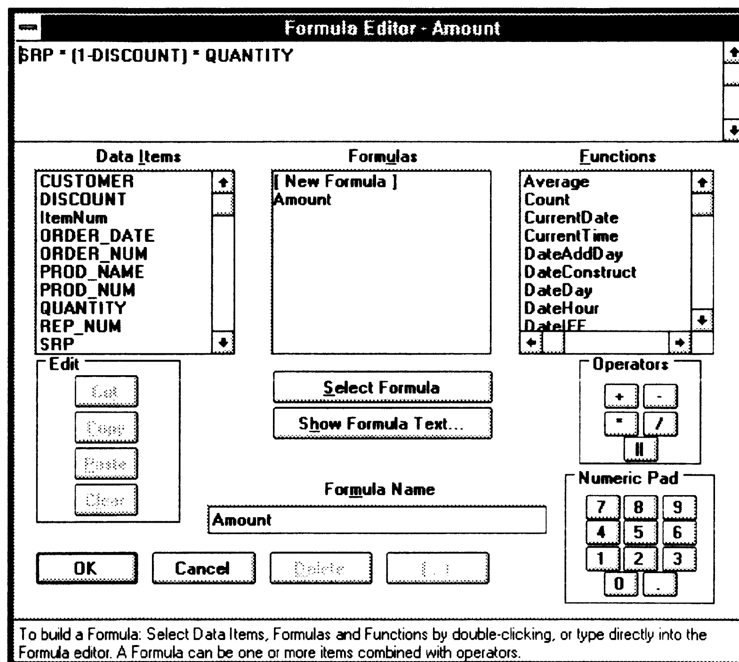

Figure 7.8 Defining a formula—Amount.

etc. yourself, you can do so. Usually it is a good idea to place them in the formula by double-clicking or clicking them as there are no chances of spelling errors.

Formula Name

After having defined the formula: SRP * (1-DISCOUNT) * QUANTITY, you can give it a meaningful name such as Amount by typing it in the Formula Name

datafield. Pressing the OK push button saves the definition of the formula and brings you back to where you were prior to entering the formula editor.

The Tool Palette

Figure 7.9 shows the tool palette. Using this tool palette, you can place objects such as boxes, fields, background text, etc. on lines of your report template. You can select any tool from the Tools menu, the Tool Palette, or by clicking the right mouse button until you see the desired tool.

Selector Tool

You can use this tool to select existing objects in your report template. Double-

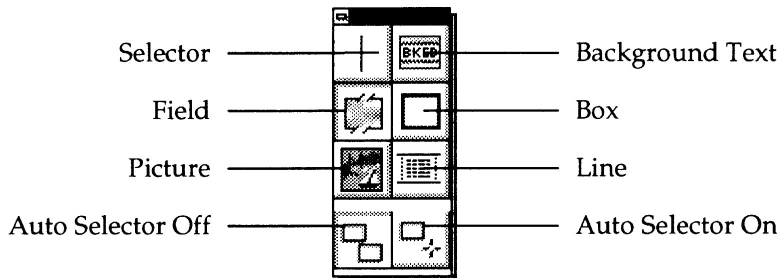

Figure 7.9 The tool palette.

click on an object or a line to bring up a dialog box to format the object or line.

Auto Selector

You can turn Auto Selector on or off. If the Auto Selector is on, ReportWindows reverts to the selector tool after each addition of a report objects to your report template. This is useful when you want to format each object immediately after adding it to the report template. If auto selector is off, the current tool remains active until another tool is selected.

Field Tool

Use this tool to draw fields in your report template. Fields can contain the following:

Possible Contents of a Field
Background text
Input items
Input variables
Input cross tabs
Input totals
Formulas

To format your field, just double-click on any field to open the Format Fields dialog box. You can also format fields from the toolbar.

Picture Tool

Use this tool to draw the picture objects in your report template. Picture objects can contain OLE objects, as well as graphics and long text that was imported as an OLE object.

Once you draw your picture object, you can re-size it and move it to any location within the same block group. Double-click on the picture object to bring up the Format Picture dialog box, where you can assign graphics, input items, or input variables as its contents. You can also set the dimensions of the picture object, and the thickness of its frame from this dialog box.

Background Text Tool

Use this tool to enter background text in your report template. Background text is any kind of header or description you want to include in your report template. Once you draw the background text you can directly enter your text. The text does not have to be enclosed by quotes.

Double-clicking on background text opens the Format Background Text dialog box. You can also format your background text directly from the Report Ruler.

Box Tool

Use the box tool to draw a box in your report template. You can use the box tool to draw a box around one or more fields, around an entire report block, or around a group of lines.

Once a box has been created, you can re-size it, or move it to any location in your report template with the Edit Cut and Edit Paste commands. You can also drag the box to a new location within the same block group. Double-clicking on any box displays the Format Box dialog box where you can set the box dimensions and border width.

Line Tool

Use the Line tool to insert blank lines into your report template. There are three ways you can insert lines with ReportWindows:

- Using the Line tool from the Tools menu.
- Using the Insert Line command from the Tools menu.
- Pressing the Insert key on the keyboard.

Once you create your line, you can use the Format Line command or double-click on the line to apply the following features to your line:

- Page break before.
- Column break before.
- Suppress blank line.
- Set the minimum height for a line.
- Add blank lines to the bottom of a block.
- Apply line borders.
- Specify colors.
- Print only when certain conditions are fulfilled.

Page Header

Figure 7.10 shows the report header and the page header for INVOICE.QRP. Note that the report header has no lines. The page header has 4 lines. The first and third lines have background text. I have stretched the background objects all the way from the left to the right and chosen center justification.

To print a double-line below the line containing INVOICE background text, double-click on the *line* which brings up the Format Line dialog box. Press the Borders... push button and from the Format dialog box, specify Double for the Bottom combo box.

Figure 7.10 The report header and page header.

When there is no report header, the page header prints even for the first page. Still, to make sure that the page header prints for the first page, you can double-click the Page Header line and check the Use With Report Header check box in the Format Block: Page Header dialog box.

Break Group Header for ORDER_NUM

Figure 7.11 shows the header for the break group for ORDER_NUM. The lines of this block are printed when the ORDER_NUM for the current row being fed to the report is different from the one for the previous row.

Figure 7.11 Break Group Header for ORDER_NUM

Since I each invoice begins on a new page, I double-clicked on the first line and chose Page Break Before from the format dialog box.

As you can see, there are 7 lines in this block. I have placed 5 fields and placed the input items for CUSTOMER, ORDER_NUM, ORDER_DATE, REP_NUM, and TERMS by first placing the fields and then choosing the respective input items from the Content combo box in the upper left corner of the ReportWindows. You can choose appropriate format, justification, font enhancement etc. either by double-clicking the field or from the toolbar. Note that I have chosen the format #0 for the field for ORDER_NUM. The rest of the objects are background text objects.

Detail Block

Figure 7.12 shows the detail block—this block is printed for each row of data fed to the report.

Detail Block
ItemN PROD_NU PROD_NAME QUANTITY SRP DISCOUNT Amount()

Figure 7.12 The Detail Block

The detail block contains only 1 line. It has several fields which contain the ItemNum input total defined earlier, input items PROD_NUM, PROD_NAME, QUANTITY, SRP, DISCOUNT, and the Amount() formula defined earlier. You can use appropriate justification and formats for various fields.

Alternate Background

As you can see in Figure 7.2, the lines for each product ordered have alternate light gray and white backgrounds. This makes it easy to read the report generated. To achieve this effect, double-click on the line to bring up the Format Line dialog box, press the Colors... push button and choose Lt Gray from the Background combo box. Finally, check the Alternate Background check box as shown in Figure 7.13.

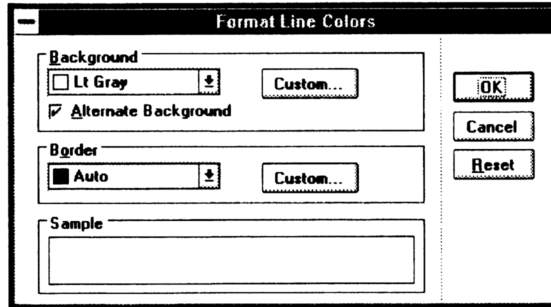

Figure 7.13 Specifying alternate background for detail block.

Break Group Footer for ORDER_NUM

Figure 7.14 shows the footer for the ORDER_NUM break group. Note that I have placed the input total TotalAmount defined earlier here. By the time this footer is printed, the input total has calculated the sum of Amount() for all of the products ordered for this ORDER_NUM.

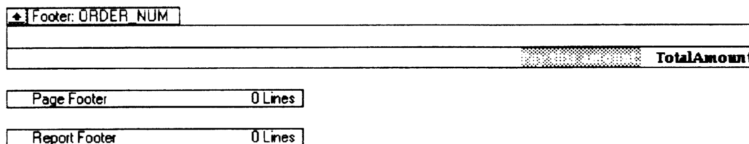

Figure 7.14 Break group footer for ORDER_NUM.

In this report template, the page footer and the report footer do not contain anything.

Printing or Displaying a Report

Now that the report template INVOICE.QRP has been designed, I explain how you can actually print or display this report from a SQLWindows application. INVOICE.APP contains the necessary code to achieve this. After displaying the usual login dialog box, the application creates the dialog box shown in Figure 7.15. The Customer combo box displays a list of the customers. It prepares this list by selecting all the CUSTOMERs from the ORDER_MASTER table. Once a user selects a customer from this list, the Order Number combo box is populated

with the ORDER_NUMs for this customer, again, from the ORDER_MASTER table.

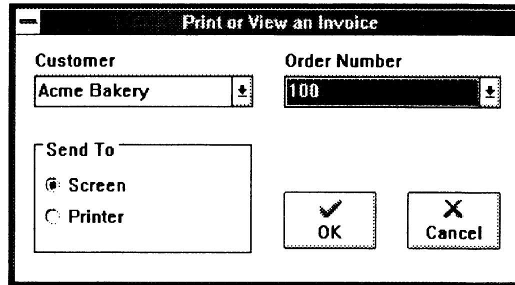

Figure 7.15 The dialog box displayed by INVOICE.APP.

The user can choose whether the report is sent to a printer or to the screen for a preview. Pressing the OK push button generates this report. After the report has been sent to the printer or the user has finished looking at the report on the screen, the user can press the Cancel push button to close the dialog box and leave the application.

Let me show you the code behind this.

Displaying the Report Dialog Box

INVOICE.APP defines a programmer message constant PM_Populate. This message is sent to the combo boxes. It also defines some global variables, including a sql handle hSqlReport which is connected by the login dialog box. If the login succeeds, the application creates the report dialog box by calling SalModalDialog.

```
Application Description: INVOICE.APP
  Chapter 7
  Generating Reports
  Power Programming with SQLWindows
  by Rajesh Lalwani.
  Copyright (c) 1994 by Gupta Corporation.
  All rights reserved.
 Global Declarations
  Constants
   User
    Number: PM_Populate = SAM_User
```

```
    Number: ERROR_TIMEOUT = -1805
    Number: ERROR_ROWID = 806
  Variables
   Sql Handle: hSqlReport
   Boolean: bConnectReport
   Sql Handle: hSqlError
   Number: nError
   Number: nPos
   Boolean: bRollback
   String: strMessage
  Application Actions
   On SAM_AppStartup
    ! Login Dialog Box with defaults for database,
      user, password. Pass hSqlReport as the receive
      parameter.
    Set bConnectReport = SalModalDialog( dlgLogin, hWndNULL,
      'GUPTA', 'SYSADM', 'SYSADM', hSqlReport )
    ! Quit the application if connect not successful
    If not bConnectReport
     Call SalQuit()
    ! Connect successful, create the dialog box dlgReport
    Call SalModalDialog( dlgReport, hWndNULL )
   On SAM_AppExit
    ! Disconnect the sql handle of our first connect
    If bConnectReport
     Call SqlDisconnect( hSqlReport )
   On SAM_SqlError
```

Listing 7.1 Constants, variables, and application actions for INVOICE.APP.

Populating Combo Boxes of the Report Dialog Box

The report template INVOICE.QRP can be used to generate several invoices. In case of multiple invoices each invoice, one for each order, is printed on a new page. However, in this application, only one invoice is generated. This is for a specific order selected by the user.

The report dialog box displays two combo boxes. The first one contains a list of all the customers. When a user selects a customer from this combo box, the second combo box is populated with all the orders placed by the selected customer. The user can also choose whether to send the report to the printer or to the screen. When the user presses the OK push button, the dialog box launches a report to print or display the invoice for the selected order.

Combo Box

A combo box contains a data field and a list box. The list box contains predefined scrollable items that a user chooses to fill the data field.

The list box part of a combo box can have these features:

- Sorted items.
- Vertical scroll bar.
- Can always be dropped.

The data field part of a combo box can be editable or non-editable. If the data field is non-editable, there is no space between the right side of the data field and the down arrow; if the data field is editable, there is a space between the right side of the data field and the down arrow.

One or no items in a combo box list are selected at any given time.

A combo box works as follows:

- Click an item in the list box to select it, put it in the data field part of the combo box, and close the list box.
- The arrow keys scroll the list box, change the selection, and the contents of the data field part of the combo box. If the list box is not down, the arrow keys change the selection in the data field.
- If the combo box is editable, press a key to scroll to an item that starts with that letter.
- Alt+up arrow and Alt+down arrow open and close the list box.

As you can see from Listing 7.2, I use SalListPopulate to populate the combo boxes.

SalListPopulate

```
bOk = SalListPopulate ( hWndList, hSql, strSelect )
```

SalListPopulate populates a list box or combo box with a result set.

If the SELECT statement returns data from multiple columns, each column's data displayed in a list box is separated by tabs. However, due to a Microsoft

Windows limitation, each column's data displayed in a combo box is separated by a single '|' character. There is no space between one column's data, the separator character, and another column's data.

hWndList is the window handle (or name) of the list box or combo box to populate.

hSql is the sql handle of a SELECT statement.

strSelect contains the SELECT statement. The SELECT statement can contain bind variables, but it cannot contain INTO variables. If strSelect is null (''), SQLWindows uses the previously prepared SELECT statement associated with hSql. This avoids repeated preparation each time SalListPopulate executes.

bOk is TRUE if the function succeeds and FALSE if any of the parameters are invalid or if strSelect contains INTO variables.

In this application, I populate the Customer combo box with 'SELECT DISTINCT CUSTOMER FROM ORDER_MASTER'. Notice the use of the DISTINCT keyword. It makes sure that each customer appears only once in the combo box even though ORDER_MASTER may contain several orders from this customer.

SAM_Click

SAM_Click is sent to a combo box when you click on an entry in the drop-down list box or perform a keyboard activity which simulates a mouse click. If you click the data field portion of the combo box or the button that invokes the drop-down list box, no SAM_Click is sent.

SAM_Click is sent to a list box when you click on an entry or perform a keyboard activity which simulates a mouse click. If you click on an empty list box or click on a portion of the list box that contains no entries, no SAM_Click is sent.

As you can see in Listing 7.2, cmbCustomer processes SAM_Click, makes sure that the user has chosen something from the list, and posts a PM_Populate message to cmbOrderNum. Upon receiving this message, cmbOrderNum populates itself with 'SELECT ORDER_NUM FROM ORDER_MASTER WHERE CUSTOMER = :cmbCustomer' by calling SalListPopulate. Notice the use :cmbCustomer as a bind variable. When the name of the combo box is used like a variable, it refers to the data field portion of the combo box.

The data type of a combo box can only be a String (or Long String). This is the reason why cmbOrderNum converts an order number such as '101' to a number 101 by calling SalStrToNumber.

```
Dialog Box: dlgReport
 Title: Print or View an Invoice
 Description:
   This dialog box displays all the customers
   in a combo box. When a user selects a
   customer, another combo box displays all the
   orders from that customer. After choosing the
   order number and destination of the report (screen or
   printer), OK will send the report. Cancel ends
   the dialog box.
 Contents
  Combo Box: cmbCustomer
   Editable? No
   String Type: String
   Maximum Data Length: 50
   Sorted? Yes
   Always Show List? No
   Vertical Scroll? Yes
   Message Actions
    On SAM_Create
     Call SalListPopulate( hWndItem, hSqlReporst,
       'SELECT DISTINCT CUSTOMER FROM ORDER_MASTER' )
    On SAM_Click
     If not SalIsNull( hWndItem )
      ! Now ask cmbOrderNum to populate itself
        with the order numbers from this customer.
      Call SalPostMsg( cmbOrderNum, PM_Populate, 0, 0 )
  Combo Box: cmbOrderNum
   Editable? No
   String Type: String
   Maximum Data Length: Default
   Sorted? Yes
   Always Show List? No
   Vertical Scroll? Yes
   Message Actions
    On SAM_Create
     Call SalDisableWindow( hWndItem )
    On PM_Populate
     Call SalEnableWindow( hWndItem )
     Call SalListPopulate( hWndItem, hSqlReport,
       'SELECT ORDER_NUM FROM ORDER_MASTER
```

```
        WHERE CUSTOMER = :cmbCustomer' )
     On SAM_Click
      If not SalIsNull( hWndItem )
       ! Store the string cmbOrderNum into a Number variable.
       Set nOrderNum = SalStrToNumber( cmbOrderNum )
       ! Now enable the OK push button
       Call SalEnableWindow( pbOk )
   Background Text: Customer
   Background Text: Order Number
   Group Box: Send To
   Radio Button: rbScreen
    Title: Screen
   Radio Button: rbPrinter
    Title: Printer
   Pushbutton: pbOk
   Pushbutton: pbCancel
    Title: Cancel
    Message Actions
     On SAM_Click
      Call SalEndDialog( hWndForm, TRUE )
  Window Variables
   Number: nOrderNum
   Date/Time: dtOrderDate
   String: strTerms
   String: strRepNum
   Number: nProdNum
   Number: nQuantity
   Number: nDiscount
   String: strProdName
   Number: nSRP
   Window Handle: hWndReport
   Number: nErr
   Number: nReturn
```

Listing 7.2 Populating the combo boxes of the dialog box.

Launching the Report

As you can see in Listing 7.3, pbOk is responsible for launching the report. Upon receiving SAM_Click, it checks which of the two radio buttons, rbPrinter and rbScreen, is selected by the user and calls SalReportPrint or SalReportView respectively.

SalReportPrint

```
hWndReport= SalReportPrint ( hWndFrm, strTemplate,
            strVariables, strInputs, nCopies, nOptions,
            nFirstPage, nLastPage, nErr )
```

SalReportPrint prints a report. SalReportPrint creates a minimized ReportWindows window. A print status dialog box displays while printing. When the print job completes, ReportWindows automatically ends.

This function returns before printing begins.

hWndFrm is the window handle (or name) of the application window that processes SAM_Report* messages.

strTemplate is the report template name.

strVariables is a string containing a comma-separated list of SQLWindows variables from which to fetch data. The data types of these variables must match the data types of the input names in strInputs.

strInputs is a string containing a comma-separated list of the report's input names. The contents of this string are case-sensitive and must match the case of the input names declared in the report template.

nCopies is the number of copies. If you specify a negative number, SQLWindows prints one copy.

nOptions is a number which specifies the options. You can combine two or more of these constants with the OR (|) operator:

Constant to be used with SalReportPrint or SalReportPrintToFile.	Description
RPT_PrintAll	Print all pages of a report.
RPT_PrintDraft	Print a report in draft mode.
RPT_PrintNoAbort	SQLWindows does not display the dialog box that lets the user cancel the report.

Constant to be used with SalReportPrint or SalReportPrintToFile.	Description
RPT_PrintNoErrors	Suppress error message dialog boxes during printing.
RPT_PrintNoWarn	Suppress warnings about margin overflow and tiled pages.
RPT_PrintRange	Print a range of pages. You must also set the values of the first (nFirstPage) and last (nLastPage) pages of the report.

nFirstPage is the first page of the report (if nOptions includes RPT_PrintRange).

nLastPage is the last page of the report (if nOptions includes RPT_PrintRange).

nErr is a receive number. If an error occurs, this parameter is equal to one of the RPT_Err* values. In run mode at designtime, SQLWindows always displays a dialog box with an explanation if an error happens. For *.RUN and *.EXE applications, SQLWindows only displays an error dialog box if you set this parameter to 1 before calling SalReportPrint.

hWndReport is the handle of the report window if the function succeeds.

SalReportView

```
hWndReport= SalReportView ( hWndFrm, hWndRptTemp,
            strTemplate, strVariables, strInputs, nErr )
```

SalReportView displays a report in preview mode. This function returns before ReportWindows displays the report.

Parameters hWndFrm, strTemplate, strVariables, strInputs, and nErr have the same meaning as for SalReportPrint (see above).

hWndRptTemp is an optional window handle (or name) to a custom report window template. If null, ReportWindows creates its own window in which it displays the report.

hWndReport is the handle of the report window if the function succeeds. SQLWindows returns the window handle before ReportWindows displays the report.

```
Pushbutton: pbOk
 Title: OK
 Message Actions
  On SAM_Create
   Call SalDisableWindow( hWndItem )
  On SAM_Click
   Call SalWaitCursor( TRUE )
   ! For *.RUN and *.EXE applications, SQLWindows only displays
     an error dialog box if you set nErr parameter to 1 before
     calling SalReportPrint or SalReportView.
   Set nErr = 1
   If rbPrinter
    ! Send the report to the printer
    Set hWndReport = SalReportPrint( hWndForm,
      'INVOICE.QRP',
      ':nOrderNum, :dtOrderDate, :strTerms, :strRepNum,
      :cmbCustomer, :nProdNum, :nQuantity, :nDiscount,
      :strProdName, :nSRP',
      'ORDER_NUM, ORDER_DATE, TERMS, REP_NUM,
      CUSTOMER, PROD_NUM, QUANTITY, DISCOUNT,
      PROD_NAME, SRP',
      1, RPT_PrintAll, 0, 0, nErr )
   Else
    ! Send the report to the screen
    Set hWndReport = SalReportView( hWndForm, hWndNULL,
      'INVOICE.QRP',
      ':nOrderNum, :dtOrderDate, :strTerms, :strRepNum,
      :cmbCustomer, :nProdNum, :nQuantity, :nDiscount,
      :strProdName, :nSRP',
      'ORDER_NUM, ORDER_DATE, TERMS, REP_NUM,
      CUSTOMER, PROD_NUM, QUANTITY, DISCOUNT,
      PROD_NAME, SRP', nErr )
```

Listing 7.3 pbOk launching the report.

Feeding Data to the Report – One Row at a Time

Figure 7.16 shows how a SQLWindows application and ReportWindows interact with each other at runtime. A SQLWindows application communicates with ReportWindows by calling SalReport* functions. ReportWindows communicates with the SQLWindows application through SAM_Report* messages. The

SQLWindows application is a server because it supplies data whereas ReportWindows is a client because it requests data.

SAM_ReportStart

SAM_ReportStart is sent after the application calls the report with either SalReportView or SalReportPrint to indicate that the report is starting. SAM_ReportStart is sent before the report displays or prints.

You can use SAM_ReportStart to do any initialization necessary before sending data to the report.

SQLWindows ignores SAM_ReportStart's return value.

wParam contains the window handle of the window that started the report.

In this application, since the sql handle is already connected, in response to SAM_ReportStart, I simply prepare the following SQL statement:

```
'SELECT ORDER_DATE, TERMS, REP_NUM, ORDER_DETAIL.PROD_NUM,
QUANTITY, DISCOUNT, PROD_NAME, SRP
INTO :dtOrderDate, :strTerms, :strRepNum, :nProdNum,
:nQuantity, :nDiscount, :strProdName, :nSRP
FROM ORDER_MASTER, ORDER_DETAIL, PRODUCT
WHERE ORDER_MASTER.ORDER_NUM = :cmbOrderNum AND
ORDER_MASTER.ORDER_NUM = ORDER_DETAIL.ORDER_NUM AND
ORDER_DETAIL.PROD_NUM = PRODUCT.PROD_NUM'
```

SAM_ReportFetchInit

This message is sent when ReportWindows is ready to format the first page of a report. ReportWindows sends SAM_ReportFetchInit after sending SAM_ReportStart. SAM_ReportFetchInit means that ReportWindows is ready to receive data from the application.

If you Return FALSE, the report stops. If you do not Return a value, or you do not process the message, the report continues.

wParam is the window handle of a ReportWindows window.

In this application, in response to SAM_ReportFetchInit, I call SqlExecute to execute the SELECT statement. If there is some error, I return FALSE so that the report stops.

ReportWindows may send SAM_ReportFetchInit more than once depending on how the user scrolls through the report pages, whether the user chooses to print the report after previewing it on the screen, or if pre-process (two pass) totals are defined in the report. In such situations, ReportWindows has to fetch the first row of data more than once.

It is very important that you do *not* call SqlExecute in response to SAM_ReportStart. Calling SqlExecute in response to a SAM_ReportFetchInit message correctly executes the SQL statement and hence next call to SqlFetchNext correctly fetches the first row. If you call SqlExecute in response to a SAM_ReportStart message, which is sent only once in the beginning, the needed initialization is not done when SAM_ReportFetchInit is received.

SAM_ReportFinish

SAM_ReportFinish is sent when the report finishes. The application can use this message to do any cleanup that is needed following the generation of a report.

wParam is the window handle of the window that started the report.

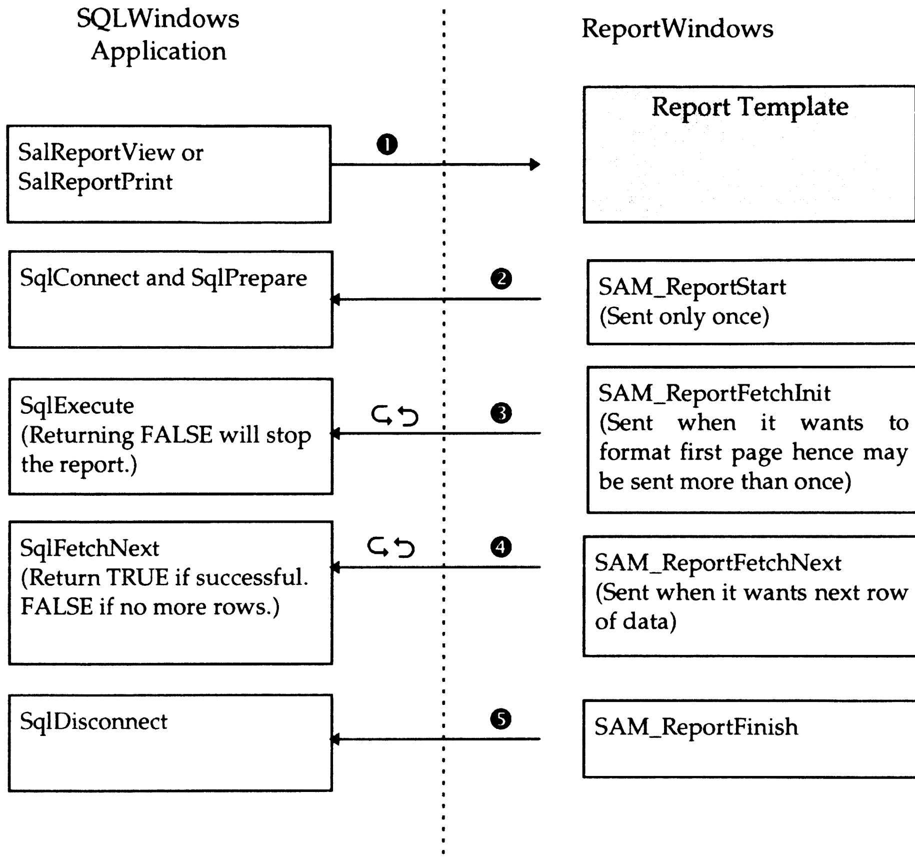

Figure 7.16 Interaction between a SQLWindows application and ReportWindows at runtime. Actions of the SQLWindows application are suggestions only. It is assumed the data is coming from a database.

```
Message Actions
 On SAM_ReportStart
  If not SqlPrepare( hSqlReport,
    'SELECT ORDER_DATE, TERMS, REP_NUM,
    ORDER_DETAIL.PROD_NUM, QUANTITY, DISCOUNT, PROD_NAME, SRP
    INTO :dtOrderDate, :strTerms, :strRepNum,
    :nProdNum, :nQuantity, :nDiscount, :strProdName, :nSRP
    FROM ORDER_MASTER, ORDER_DETAIL, PRODUCT
    WHERE ORDER_MASTER.ORDER_NUM = :cmbOrderNum AND
    ORDER_MASTER.ORDER_NUM = ORDER_DETAIL.ORDER_NUM AND
    ORDER_DETAIL.PROD_NUM = PRODUCT.PROD_NUM')
   Call SalMessageBeep( 0 )
   Call SalMessageBox( 'Failed to prepare the SQL statement.',
     'Error', MB_Ok | MB_IconExclamation )
 On SAM_ReportFetchInit
  If not SqlExecute( hSqlReport )
   Call SalMessageBeep( 0 )
   Call SalMessageBox(
     'Failed to execute the SQL statement. Exiting.',
     'Error', MB_Ok | MB_IconExclamation )
   Return FALSE
 On SAM_ReportFetchNext
  Return SqlFetchNext( hSqlReport, nReturn )
 On SAM_ReportFinish
  Call SalWaitCursor( FALSE )
```

Listing 7.4 Handling report messages.

Cross Tabular Report

SALE.APP generates a cross tabular report. It uses a report template SALE.QRP which defines a cross tab. Figure 7.17 shows what a cross tabular report looks like.

	1	2	3	4	Total
Ann Fletcher		$89,125.00	$95,275.00	$43,250.00	$227,650.00
Bob Gomez	$23,650.00	$78,800.00	$119,000.00	$32,000.00	$253,450.00
Fred Jones	$96,550.00		$42,950.00	$52,200.00	$191,700.00
Hugh Hobart	$53,650.00	$56,375.00	$53,100.00		$163,125.00
Mary Mullen	$87,675.00	$65,450.00	$66,300.00	$43,500.00	$262,925.00
Total	$261,525.00	$289,750.00	$376,625.00	$170,950.00	$1,098,850.00

Figure 7.17 A cross tabular report showing revenues earned by each salesperson in each quarter.

This cross tabular report has been generated from the rows of the MYSALES table from the GUPTA database. The MYSALES table has three columns: SALESPERSO (VARCHAR 20), QUARTER (FLOAT), and AMOUNT (FLOAT). Let me show you all rows corresponding to Ann Fletcher in the MYSALES table.

SALESPERSO	QUARTER	AMOUNT
Ann Fletcher	2	150
Ann Fletcher	2	750
Ann Fletcher	2	800
Ann Fletcher	2	5,200
Ann Fletcher	2	5,600
Ann Fletcher	2	7,375
Ann Fletcher	2	21,625
Ann Fletcher	2	22,125
Ann Fletcher	2	25,500
Ann Fletcher	3	200
Ann Fletcher	3	400
Ann Fletcher	3	1,050
Ann Fletcher	3	5,800
Ann Fletcher	3	6,375
Ann Fletcher	3	6,875
Ann Fletcher	3	22,375
Ann Fletcher	3	25,800
Ann Fletcher	3	26,400
Ann Fletcher	4	21,375
Ann Fletcher	4	21,875

Figure 7.18 All rows corresponding to Ann Fletcher in the MYSALES table.

As you can see from the cross tabular report generated, ReportWindows is able to sum the AMOUNTs for each sales person and for each quarter. Sales persons form the rows of the cross tab, quarters form the columns of the cross tab, and the sums of the AMOUNTs form the cells.

You can also see that the report generated contains the summary information for each sales person (for all the quarters) as well as the summary information for each quarter (for all the sales persons). It also has the total for the entire cross tab.

Finally, notice 1, 2, 3, and 4 as headings for the columns and the names of the sales persons as headings for the rows.

You do all this by defining a single cross tab in the report template.

Defining Input Items

As explained before, define the following input items by choosing Format, Input, Input Items... from the menu: SALESPERSO (String), QUARTER (Number), and AMOUNT (Number).

Defining Input Cross Tab

To create a new cross tab, choose Format, Input, Crosstabs... from the menu of ReportWindows. Make sure the Quick Crosstab radio button of the Format Crosstabs dialog box is selected. Press the Create New Crosstab... push button to create a new crosstab. ReportWindows guides you through the steps to define rows, columns, etc.

Defining Rows

The Define Rows dialog box displays the input items of the report template in the Categories list box. Since we want SALESPERSO to form the rows, select SALESPERSO and copy it to the Selected Row Categories list box by pressing the right arrow push button. Make sure the Sort Order is Ascending and the check boxes Show and Suppress Repeating in the Heading Options group box are checked.

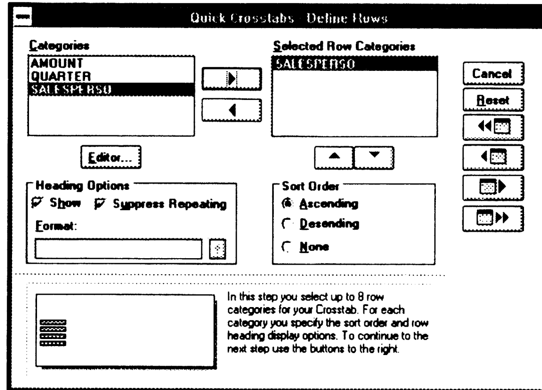

Figure 7.19 Defining rows of the cross tab.

Defining Columns

Once you are done with this step of defining rows, press the right arrow push button on the right hand side of the Define Rows dialog box to go to the second step of defining columns.

Since we want QUARTERs to form the columns of the cross tab, move QUARTER from the Categories list box to the Selected Column Categories list box. Make sure the Sort Order is Ascending and the check boxes Show and Suppress Repeating in the Heading Options group box are checked.

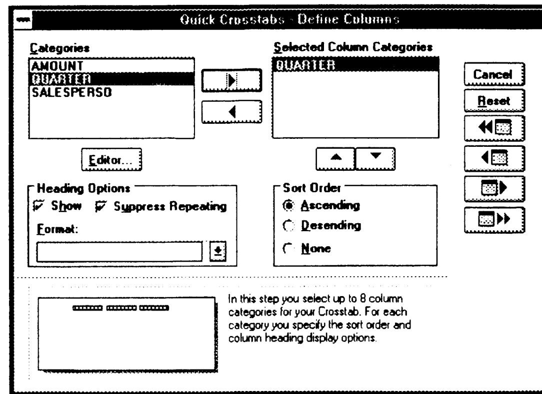

Figure 7.20 Defining columns of the cross tab.

Defining Cell Value

Once you are done with the step of defining the columns, press the right arrow push button on the right hand side of the Define Columns dialog box to go to the third step of defining the cell value.

Since we want to total the AMOUNTs, copy AMOUNT from the Values list box to the Cell Value list box. In the next step, you define what statistics (in this case, sum) you want to perform on this cell.

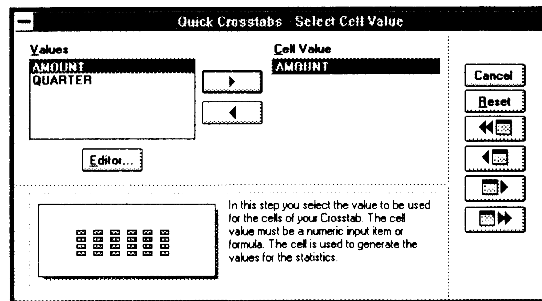

Figure 7.21 Defining cell value of the cross tab.

Defining Statistics

Press the right arrow push button on the right hand side of the Define Cell Value dialog box to go to the fourth step of defining statistics for the cell value.

Since we want to total the AMOUNTs, copy Sum from the Statistics list box to the Selected list box. This cross tab performs only the total of the AMOUNTs. It is, however, possible to perform multiple statistics such as Sum, Minimum, and Maximum on the cell value. Choose $ as the format, uncheck the Default Headings check box, and delete the default heading Sum from the Heading data field.

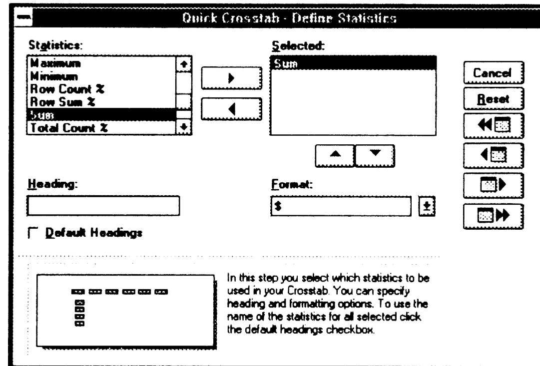

Figure 7.22 Defining statistics for the cell value of the cross tab.

Defining Summary Statistics

Press the right arrow push button on the right hand side of the Define Statistics dialog box to go to the fifth step of defining summary statistics.

Since we want the total for each salesperson, quarter, and the whole cross tab, check the Columns, Rows, and Matrix check boxes for Summary Statistics. Specify Total as the headings for both column and row summary headings.

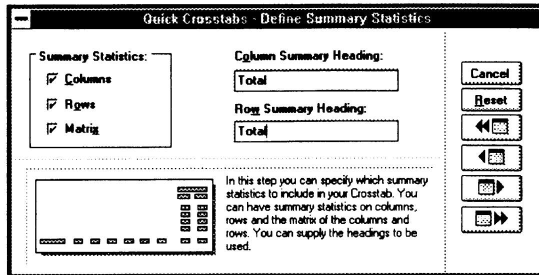

Figure 7.23 Defining summary statistics of the cross tab.

Naming the Cross Tab and Specifying the Restart Event

Press the right arrow push button on the right hand side of the Define Summary Statistics dialog box to go to the last step of defining the restart event and saving the cross tab with a name.

Specify xTabSalesActivity as the Crosstab Name, and First Fetch as the Restart Event. The restart event for a cross tab is similar to the restart event of input totals as I explained earlier. Choose Close from the Format Crosstabs dialog box.

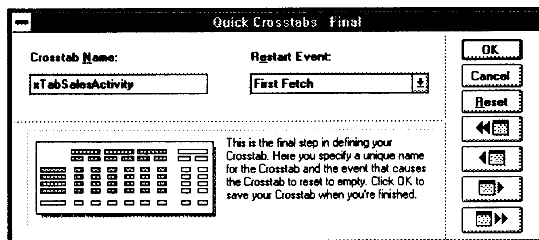

Figure 7.24 Defining restart event and giving a name to the cross tab.

Placing the Cross Tab in the Report Template

Add a couple of lines in the Page Header block and place background text for the title to be printed on top of a page.

Add a new line in the Report Footer block, place a field on the entire width of the line, and place the cross tab xTabSalesActivity by choosing it from the Content combo box in the toolbar.

ReportWindows builds the cross tab as each row of data is fed to the report at runtime. Since we are only interested in the final cross tab and not in the intermediate snap shots, it is important that you put xTabSalesActivity in the Report Footer. If you place the cross tab, for example, in the Detail Block, you can see the cross tab as it is being built. Be careful before you print such a report, because it can be very large. It prints entire cross tabs for each row in the MYSALES table.

Report Header 0 Lines

Page Header

Sales Report by Sales Person and Quarter

Detail Block 0 Lines

Page Footer 0 Lines

Report Footer

xTabSalesActivity

Figure 7.25 Defining Page Header and placing xTabSalesActivity in the Report Footer.

Fine Tuning Tabs and Choosing Landscape Orientation

You may find the columns of the cross tab too close to each other. To change the distance, change the tab settings for the field containing xTabSalesActivity. To do so, select the field, and choose Format, Tabs... from the menu. You see the Format Tab Settings dialog box:

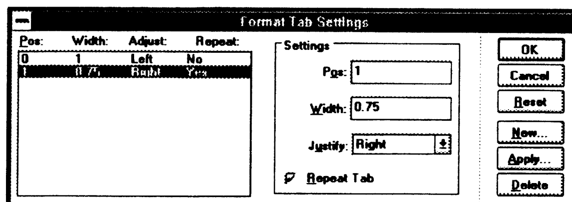

Figure 7.26 Modifying tab settings for the field containing the cross tab.

Notice that the first tab is for the row headings and the second one is for the columns and is a repeat tab, i.e., it repeats for each of the columns.

Finally, if you want, you can generate the report in the landscape orientation. To do this, choose Format, Report... from the menu and select Landscape radio button for the orientation.

Feeding Data to the Report

Launching the report and feeding data to the report at runtime is very similar to INVOICE.APP as explained earlier. You can see the necessary code in SALE.APP. Here is the SELECT statement I use to get the data from the MYSALES table of GUPTA database:

```
'SELECT SALESPERSO, QUARTER, AMOUNT INTO :strSalesPerson,
:nQuarter, :nAmount FROM MYSALES'
```

Mailing Labels

In Chapter 5, I had developed PHAD.APP. This application manages PHone numbers and ADdresses. It provides three reports—normal, two columns, and mailing labels. Let me explain how I created the report template PHADMLBL.QRP for mailing labels.

The report generates mailing labels which are 2-5/8" x 1". There are three columns of 10 labels per column. These mailing labels can be directly printed on or copied to Avery Products 5160 and 5260.

Defining Input Items

As explained earlier, define the input items by choosing Format, Input, Input Items... from the menu of ReportWindows. Although this report template does not use most of the input items defined, I defined the same set of input items for all three report templates for the sake of simplicity and easy maintenance of the code. Here are the input items for the report template. All of them are of the data type String: LASTNAME, FIRSTNAME, MRMS, MIDDLEINITIAL, COMPANY, HOMEPHONE, WORKPHONE, ALTERNATEPHONE, STREET1, STREET2, STREET3, CITY, STATE, ZIP, COUNTRY, OTHEROCCASION, BIRTHDAY, ANNIVERSARY, OTHERDAY, and NOTES.

Defining Number of Columns

You can define the number of columns, spacing between columns, margins, etc. by choosing Format, Report... from the menu. As shown in Figure 7.27, define the page size as Letter (8.5"x11"), orientation as Portrait, number of columns as 3, spacing between columns as 0.15625". The page margins should be 0.5" for top and bottom, and 0.15625" for left and right.

After you press the OK push button, you will see that the report template contains three columns for lines in the detail block. You can place any fields, background text, etc. on the first column only. The other columns are automatically the exact copies of this column.

Figure 7.27 Defining the number of columns, spacing between them and margins.

Placing Input Items and Formulas in Fields

Make sure there are 6 lines in the detail block as shown in Figure 7.28. Place fields on them using the Field tool of the tool palette. The middle 4 lines contain the input items COMPANY, STREET1, STREET2, and STREET3. The first line contains a formula CompleteName() which can be defined by selecting the field

on the first line and pressing the EDITOR push button from the toolbar. The formula is defined as follows:

```
StrIFF( StrLength( MRMS ),'','',MRMS || ' ' )  ||
StrIFF( StrLength( FIRSTNAME ),'','',FIRSTNAME || ' ' )  ||
StrIFF( StrLength( MIDDLEINITIAL ),'','',MIDDLEINITIAL ||' ' )||
LASTNAME
```

Listing 7.5 Definition of formula CompleteName().

| | is the string concatenation operator. StrIFF and StrLength are ReportWindows functions which you can choose from the Functions list box of the formula editor. Basically, this long formula creates a complete name of the person, such as Mr. William Clinton, adding a blank after a component only if that component is not blank.

CityStateZipCountry() is another formula which has been defined as follows:

```
CITY || ' ' || STATE || ' ' || ZIP || ' ' || COUNTRY
```

Listing 7.6 Definition of formula CityStateZipCountry().

Unlike the formula for CompleteName(), this formula does not make an effort to remove unnecessary blanks. If you are creating mailing labels, each person should have CITY, STATE, and ZIP defined for them, so it may be unnecessary.

StrLength

`StrLength(string)`

This function returns the length of string.

StrIFF

`StrIFF(value, string1, string2, string3)`

Returns a specific string based on the value provided by the value parameter. You can specify a maximum of three strings.

value contains a number that identifies the string to be returned. If

value < 0	string1 is returned
value = 0	string2 is returned
value > 0	string3 is returned

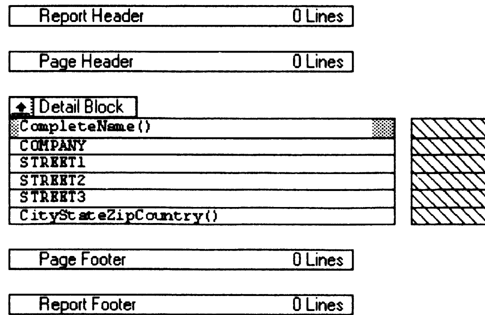

Figure 7.28 Report template for PHADMLBL.QRP.

Supressing Blank Lines

You may not have company names for some people. Also, more likely, you may not have used STREET2 or STREET3. In such cases, it is desirable that blank lines do not appear on the mailing labels. You can achieve this by double-clicking on *each* of the lines to bring up the Format Line dialog box as shown in Figure 7.29. Check the Suppress Line Spacing check box. Checking Add to Bottom of Block check box holds the printing of the blank line until the end of this block. It is not necessary to check this but this would have been necessary in the absence of another feature of Minimum Height.

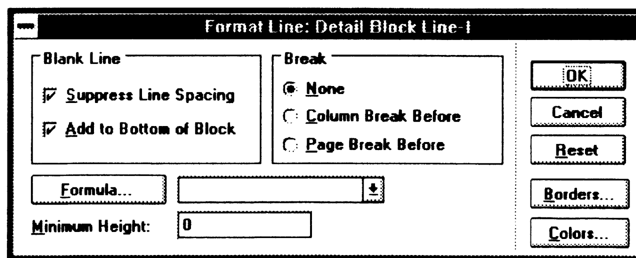

Figure 7.29 Suppressing blank lines.

Specifying Minimum Height for the Detail Block

Normally, ReportWindows prints the detail lines of the next person immediately following the detail lines of the previous person. If you do not check Add to Botton of Block check box as explained earlier, and if the previous person had two blank lines for STREET2 and STREET3 and hence suppressed, the detail lines of the next person are printed immediately following the line containing CityStateZipCountry(). Hence the mailing label of the next person will start on the mailing label of the previous person.

Even if you have checked the Add to Botton of Block check box, you do not know whether these 6 lines of the detail block are exactly 1" or not. This would depend on the font size used.

To make printing on labels of fixed height easy, ReportWindows provides a feature where you can specify the minimum height for a block. If the lines in that block do not fill the entire height of the label, it skips the rest of the height so that the next set of lines begins on the next label. You can do so by double-clicking on the Detail Block and specifying 1 (1 inch) as the minimum height in the Format Block dialog box.

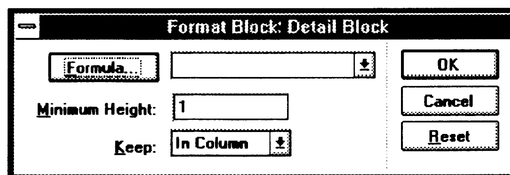

Figure 7.30 Specifying minimum height for the detail block.

Don't forget to save the report template by choosing File, Save from the menu.

Printing Multiple Reports in a Batch

There are occasions where you want to generate several reports. It would be nice to be able to check all the reports you want and press the OK push button to print all of them one after the other. This becomes particularly important if the reports are long and take a long time to print.

While there are several ways you can accomplish this, in BATCH.APP I have chosen a very simple approach to illustrate the concept. The report dialog box

presents all available reports in the form of check boxes. A user can check all, some, or none of them. When the user presses the OK push button, the OK push button sequentially looks for a check box which is checked. When it finds one, it launches the corresponding report. The dialog box handles the SAM_Report* messages. Finally, when the report is finished, the dialog box receives the SAM_ReportFinish message and, in response, unchecks this check box and posts SAM_Click message to the OK push button.

Upon receiving this SAM_Click message, the OK push button does the usual processing and looks for a checked check box. Since check boxes for reports already generated are unchecked after receiving SAM_ReportFinish, the OK push button does not launch them again. If the OK push button finds another checked check box, it launches the corresponding report and the process continues. In the end, when all the reports have been generated, the OK push button does not find any checked check box and simply changes the text of the Cancel push button to Close. Then the user can launch another set of reports or simply leave by pressing the Close push button.

Check Box

A check box, when clicked, acts like a toggle switch. Check boxes evaluate to Boolean values: they can be either On (TRUE) or Off (FALSE). More than one check box can be On at the same time.

Unlike radio buttons, check boxes are independent of each other, so any number of them can be On or Off simultaneously.

Number of Copies, All or Ranges of Pages

As you can see from the parameters of SalReportPrint, you can specify the number of copies and whether you want to print all or only a range of pages of the report. As you can see in Figure 7.31, BATCH.APP provides this option to the user. Since these options make sense only for SalReportPrint and not SalReportView, when the user selects the Printer radio button, these options are enabled but are disabled when the Screen radio button is selected. You are already familiar with SalDisableWindow and SalEnableWindow. Let me explain the two new functions – SalDisableWindowAndLabel and SalEnableWindowAndLabel.

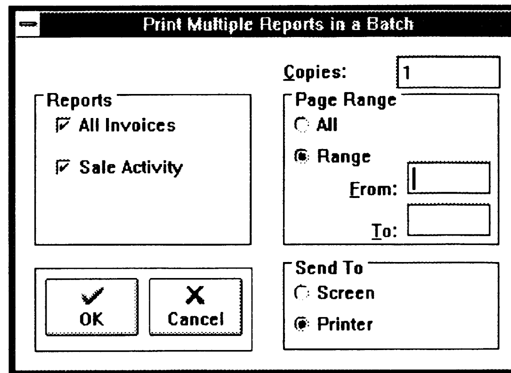

Figure 7.31 The report dialog box for generating multiple reports in a batch.

SalDisableWindowAndLabel

```
bOk = SalDisableWindowAndLabel ( hWndDisable )
```

Disables keyboard and mouse input to a window and grays out its associated label.

A label is the background text that immediately precedes the window in the outline.

If the window contains text (for example, a push button), the text is grayed. If the window is a data field, it cannot receive the focus.

hWndDisable is the window handle (or name) of the window to disable.

bOk is TRUE if the function succeeds and FALSE if it fails.

SalEnableWindowAndLabel

```
bOk = SalEnableWindowAndLabel ( hWndEnable )
```

Enables keyboard and mouse input to a window and enables its associated label as well.

A label is the background text that immediately precedes the window in the outline.

hWndEnable is the window handle (or name) of the window to enable.

bOk is TRUE if the function succeeds and FALSE if it fails.

```
Application Description: BATCH.APP
  Chapter 7
  Generating Reports
  Power Programming with SQLWindows
  by Rajesh Lalwani.
  Copyright (c) 1994 by Gupta Corporation.
  All rights reserved.
  This application lets a user select a number of reports
  by checking check boxes. When the user presses the OK push
  button, it prints all the specified reports one after the
  other. It removes check marks as reports are being printed.
 Dialog Box: dlgReport
  Title: Print Multiple Reports in a Batch
  Description:
    This dialog box presents a list of
    the reports available. A user can choose
    any and all of them by checking the option
    boxes. Since the batch mode is used
    mostly for printing, Printer is the default.
    OK sends the report. Cancel ends the dialog
    box.
  Contents
   Group Box: Reports
   Check Box: cbAllInvoices
    Title: All Invoices
   Check Box: cbSaleActivity
    Title: Sale Activity
   Group Box: Send To
   Radio Button: rbScreen
    Title: Screen
    Message Actions
     On SAM_Click
      ! Disable the options that don't make sense
      Call SalDisableWindowAndLabel( dfCopies )
      Call SalDisableWindow( rbAll )
      Call SalDisableWindow( rbRange )
      Call SalDisableWindowAndLabel( dfFrom )
      Call SalDisableWindowAndLabel( dfTo )
   Radio Button: rbPrinter
    Title: Printer
    Message Actions
     On SAM_Click
      ! Enable the options
```

```
      Call SalEnableWindowAndLabel( dfCopies )
      Call SalEnableWindow( rbAll )
      Call SalEnableWindow( rbRange )
      If rbRange
       Call SalEnableWindowAndLabel( dfFrom )
       Call SalEnableWindowAndLabel( dfTo )
   Group Box: Page Range
   Radio Button: rbAll
    Title: All
    Message Actions
     On SAM_Click
      Set nRange = RPT_PrintAll
      ! Disable the options that don't make sense
      Call SalDisableWindowAndLabel( dfFrom )
      Call SalDisableWindowAndLabel( dfTo )
   Radio Button: rbRange
    Title: Range
    Message Actions
     On SAM_Click
      Set nRange = RPT_PrintRange
      ! Enable the options
      Call SalEnableWindowAndLabel( dfFrom )
      Call SalEnableWindowAndLabel( dfTo )
   Background Text: &From:
   Data Field: dfFrom
   Background Text: &To:
   Data Field: dfTo
   Background Text: &Copies:
   Data Field: dfCopies
   Frame
   Pushbutton: pbOk
   Pushbutton: pbCancel
    Message Actions
     On SAM_Click
      Call SalEndDialog( hWndForm, TRUE )
  Window Variables
   ! Variables for Invoice report
   String: strCustomer
   Number: nOrderNum
   Date/Time: dtOrderDate
   String: strTerms
   String: strRepNum
   Number: nProdNum
   Number: nQuantity
   Number: nDiscount
```

```
String: strProdName
Number: nSRP
! Variables for Sale report
String: strSalesPerson
Number: nQuarter
Number: nAmount
! Miscellaneous variables
Window Handle: hWndReport
Number: nErr
Number: nReturn
Number: nRange
String: strTemplate
String: strVariable
String: strInput
String: strSQL
```

Listing 7.7 Dialog box dlgReport to print multiple reports in a batch.

Launching the Next Report

As I explained earlier, pbOk goes sequentially through the list of all the check boxes and looks for the checked ones. Depending on which report is to be launched, it sets up the SQL statement to be prepared and executed later.

BATCH.APP has only two reports to choose from but you can easily extend this concept to several reports.

```
Pushbutton: pbOk
 Title: OK
 Message Actions
  On SAM_Click
   Call SalWaitCursor( TRUE )
   If cbAllInvoices
    Set strTemplate = 'INVOICE.QRP'
    Set strVariable =
      ':nOrderNum, :dtOrderDate, :strTerms, :strRepNum,
      :strCustomer, :nProdNum, :nQuantity, :nDiscount,
      :strProdName, :nSRP'
    Set strInput =
      'ORDER_NUM, ORDER_DATE, TERMS, REP_NUM,
      CUSTOMER, PROD_NUM, QUANTITY, DISCOUNT,
      PROD_NAME, SRP'
    Set strSQL =
      'SELECT ORDER_MASTER.ORDER_NUM, CUSTOMER,
      ORDER_DATE, TERMS, REP_NUM,
```

```
      ORDER_DETAIL.PROD_NUM, QUANTITY, DISCOUNT,
      PROD_NAME, SRP
      INTO :nOrderNum, :strCustomer, :dtOrderDate,
      :strTerms, :strRepNum, :nProdNum, :nQuantity,
      :nDiscount, :strProdName,:nSRP
      FROM ORDER_MASTER, ORDER_DETAIL, PRODUCT
      WHERE ORDER_MASTER.ORDER_NUM = ORDER_DETAIL.ORDER_NUM
      AND ORDER_DETAIL.PROD_NUM = PRODUCT.PROD_NUM
      ORDER BY ORDER_MASTER.ORDER_NUM'
   Else If cbSaleActivity
    Set strTemplate = 'SALE.QRP'
    Set strVariable =
      ':strSalesPerson, :nQuarter, :nAmount'
    Set strInput =
      'SALESPERSO, QUARTER, AMOUNT'
    Set strSQL =
      'SELECT SALESPERSO, QUARTER, AMOUNT
      INTO :strSalesPerson, :nQuarter, :nAmount
      FROM MYSALES'
   Else
    ! Either the user did not check any
      report, or we are done with all of them.
    Call SalWaitCursor( FALSE )
    Call SalSetWindowText( pbCancel, 'Close' )
    Return TRUE
   If rbPrinter
    ! Send the report to the printer
    Set hWndReport = SalReportPrint( hWndForm,
      strTemplate, strVariable, strInput,
      dfCopies, nRange, dfFrom, dfTo, nErr )
   Else
    ! Send the report to the screen
    Set hWndReport = SalReportView( hWndForm, hWndNULL,
      strTemplate, strVariable, strInput, nErr )
```

Listing 7.8 pbOk launching a report—the next one in line.

Handling Report Messages

Upon receiving the SAM_Create message, the report dialog box sets up the default value for the number of copies. It also makes the Printer and All radio buttons the selected ones in the beginning.

SAM_ReportStart, SAM_ReportFetchInit, and SAM_ReportFetchNext all do the usual processing. Since all the reports share the same sql handle, there is no need

to take different actions based on which report is being generated. But you can easily check to see which report is being generated by looking at the check boxes and do special processing.

Notice that upon receiving SAM_ReportFinish message, I uncheck the corresponding check box before posting a SAM_Click message to pbOk again. This way the user knows that this report is finished and also pbOk does not launch it again.

```
Message Actions
 On SAM_Create
  ! Set the defaults
  ! Check all the reports
  Set cbAllInvoices = TRUE
  Set cbSaleActivity = TRUE
  ! Printer is the default destination
  Set rbPrinter = TRUE
  Call SalPostMsg( rbPrinter, SAM_Click, 0, 0 )
  ! One copy is the default
  Set dfCopies = 1
  ! Print all the pages
  Set rbAll = TRUE
  Call SalPostMsg( rbAll, SAM_Click, 0, 0 )
 On SAM_ReportStart
  If not SqlPrepare( hSqlReport, strSQL)
   Call SalMessageBeep( 0 )
   Call SalMessageBox(
     'Failed to prepare the SQL statement. Exiting.',
     'Error', MB_Ok | MB_IconExclamation )
 On SAM_ReportFetchInit
  If not SqlExecute( hSqlReport )
   Call SalMessageBeep( 0 )
   Call SalMessageBox(
     'Failed to execute the SQL statement. Exiting.',
     'Error', MB_Ok | MB_IconExclamation )
   Return FALSE
 On SAM_ReportFetchNext
  If SqlFetchNext( hSqlReport, nReturn ) and
    nReturn = FETCH_Ok
   Return TRUE
  Else
   Return FALSE
 On SAM_ReportFinish
  ! Clear the check box for the report just finished.
    The order of the If/Else statements must be same as
```

```
    the order used in the SAM_Click section of pbOk.
  If cbAllInvoices
   Set cbAllInvoices = FALSE
  Else If cbSaleActivity
   Set cbSaleActivity = FALSE
  ! Send a SAM_Click message to pbOk so that if
    there are more reports to be printed, pbOk will
    print them. Note that the check box for the
    report just finished has been cleared.
  Call SalPostMsg( pbOk, SAM_Click, 0, 0 )
```

Listing 7.9 Handling report messages.

8

Creating Your Own QuickObjects

In this chapter, I show you how to create your own QuickObjects. You can either derive new QuickObjects from existing ones and modify their behavior, or you can create new QuickObjects from scratch. In this chapter, I give you an example of each.

Creating Your Own QuickObjects

In this section, I show you how to create a push button QuickObject called cMicroHelp. When a developer includes the MCROHELP.APL library and chooses the push button tool from the tool palette, cMicroHelp is listed in the list of the available classes. See Figure 8.1.

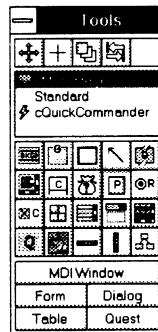

Figure 8.1 QuickObject cMicroHelp displays in the tool palette.

When the developer chooses this QuickObject and drops it on the window, a dialog box as shown in Figure 8.2 displays. Here the developer can specify the micro help to be displayed on the status bar when, at runtime, the mouse is over this push button. The developer can specify whether the micro help should be displayed on the status bar of the form window or the MDI window.

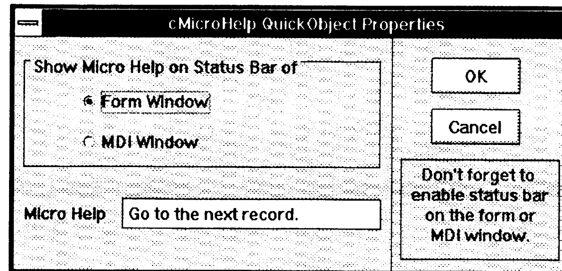

Figure 8.2 Dialog box displays when the cMicroHelp QuickObject is dropped on a window.

What is a QuickObject?

A QuickObject is a SQLWindows class that a developer configures at designtime through a custom graphical interface. This graphical interface is available directly from the SQLWindows customizer or can be launched when a developer drops an object of this class at designtime. This feature lets you customize the SQLWindows design environment to suit your own needs and makes development teams more productive.

You can make any SQLWindows visual class into a QuickObject using the QuickObject editor. The QuickObject editor lets you associate a custom graphical interface with a SQLWindows class. You implement the custom graphical interface in a standard SQLWindows application that you write. SQLWindows executes this application when a developer configures an instance of the class.

With cMicroHelp, when an instance of this QuickObject is dropped on the window or when the developer chooses cMicroHelp Properties... from the customizer, SQLWindows launches QCKPROP.EXE. I show you later how you can use the QuickObject editor to tell SQLWindows to launch this executable.

Named Properties

The concept behind a QuickObject is that you extend the designtime user interface by letting a developer specify information that is specific to the instance of the object being created or edited.

A developer specifies this information in a declarative way (via the user custom interface application) and then the object uses it at runtime.

Instance-specific information is specified in the form of named properties. As a QuickObject developer, you choose the names for properties. A QuickObject class can have as many named properties as needed. Each named property can be assigned a string value.

For example, to create a QuickObject class that shows text in both uppercase and lowercase, you associate a "CASE" property with a class that can have a value of "UPPER" or "LOWER". You use the QuickObject editor to specify a custom interface for your class that lets a developer set the "CASE" property at designtime to "UPPER" or "LOWER". At runtime, an instance of your class queries the value of the "CASE" property and changes its behavior according to the value that the developer specified at designtime. The result is that you write much less code and classes are easier to re-use because their interface is more graphical and intuitive.

Process of Creating a QuickObject

The process of creating a QuickObject consists of three steps:

1. Defining the QuickObject class. This class retrieves one or more named properties to set the values of instance variables at runtime.
2. Writing a custom interface application that lets a developer define or change the values of named properties at designtime.
3. Using the QuickObject editor to associate the QuickObject class with the custom interface application.

Defining a QuickObject Class

Listing 8.1 shows MCROHELP.APL where I define the cMicroHelp class. The class cMicroHelp is derived from cQuickObject which defines the function GetProperty.

When the push button is created, it calls GetProperty to retrieve the value of the "WNDTYPE" property. Depending on whether this property is "MDI" or "Form", it stores hWndMDI or hWndForm in the instance variable hWndMDIOrForm. This instance variable is later passed to the SalStatusSetText function. It also retrieves the property "MICROHELP" set by a developer at designtime. It stores the value of this property in the instance variable strMicroHelp.

cMicroHelp processes a Windows message WM_MOUSEMOVE. The WM_MOUSEMOVE message is sent to a window when the mouse cursor moves. Unless the mouse is captured by some other window, the message goes to the window beneath the cursor.

Upon receiving WM_MOUSEMOVE, the push button calls the function SalStatusSetText to set the status bar text of hWndMDIOrForm with strMicroHelp.

```
Application Description: MCROHELP.APL
  Creating Your Own QuickObjects
  Chapter 8
  Power Programming with SQLWindows
  by Rajesh Lalwani.
  Copyright (c) 1994 by Gupta Corporation.
  All rights reserved.
  Constants
   User
    ! Microsoft Windows constant
    Number: WM_MOUSEMOVE = 0x0200
  Class Definitions
   ! cMicroHelp Push Button QuickObject
   Pushbutton Class: cMicroHelp
    List in Tool Palette? Yes
    Property Template: QCKPROP.EXE,dlgSetProperties,cMicroHelp
      Properties...,MCROHELP.BMP,Y,N
    Description: This push button QuickObject displays the
      micro help message in the status bar when
      the mouse is over the push button.
    Derived From
     Class: cQuickObject
    Instance Variables
     String: strMicroHelp
     Window Handle: hWndMDIOrForm
    Functions
     Function: GetWindowHandle
      Description: This function returns the window
        handle of the form or the MDI window
        depending on the property "WNDTYPE"
      Returns
       Window Handle:
      Local variables
       String: strWndType
```

```
    Actions
     Call GetProperty ("WNDTYPE", strWndType)
     If strWndType = "MDI"
      Return hWndMDI
     Else
      Return hWndForm
  Message Actions
   On SAM_Create
    ! Set the values of the instance variables
      depending on the properties specified at designtime.
    Set hWndMDIOrForm = GetWindowHandle(  )
    Call GetProperty("MICROHELP", strMicroHelp)
   On WM_MOUSEMOVE
    Call SalStatusSetText( hWndMDIOrForm, strMicroHelp )
```

Listing 8.1 MCROHELP.APL defines the cMicroHelp QuickObject.

Custom Interface Applications

A Custom Interface Application is launched by SQLWindows when a developer wants to define or change the properties of a QuickObject instance. For cMicroHelp, the two properties are "WNDTYPE" and "MICROHELP" which the developer can specify using a dialog box shown in Figure 8.2.

QuickObject custom interface applications are standard SQLWindows applications that take four command line parameters:

- The name of the window to create where the developer customizes the current instance of a QuickObject class.

  It is the application's responsibility to create the window specified by the first command line parameter. In this window, you provide an intuitive interface for developers who use your class to configure an instance of the class graphically. Use developers' input to set properties that your class uses at runtime to govern its behavior.

  In case of the cMicroHelp QuickObject, dlgSetProperties (see Figure 8.2) of QCKPROP.APP is the window created for this purpose. Specify its name using the QuickObject editor.

- The identifier of the SQLWindows application outline.

This identifier is needed by dlgSetProperties to use with functions such as SalOutlineItemSetProperty.

- The handle to the instance of the object being configured.

  Again, this identifier is needed along with the outline identifier by dlgSetProperties to use with functions such as SalOutlineItemSetProperty.

- The window to activate when your application exits.

  When the custom interface application is launched, this SQLWindows session, used by the developer to design a window is suspended. In case of cMicroHelp, it is the responsibility of dlgSetProperties to activate the SQLWindows session when this dialog box ends.

QCKPROP.APP – Custom Interface Application for cMicroHelp

For cMicroHelp QuickObject, I define the custom interface in QCKPROP.APP. See Listing 8.2. The dialog box dlgSetProperties as shown in Figure 8.2 is defined in this application. QCKPROP.EXE is the executable created from this application.

This application processes the SAM_AppStartup message and converts the second (outline id) and third (item id) command line parameters into numbers. It creates the modal dialog box specified in the first command line parameter. Note that it specifies a window handle in the fourth command line parameter as the owner of the modal dialog box. When the dialog box ends, the application quits by calling the SalQuit function.

This custom interface application is first launched when a developer drops a cMicroHelp QuickObject on a window. To modify the properties of this instance later, the developer can choose cMicroHelp Properties... from the customizer. To access the properties defined earlier by the developer, the dialog box dlgSetProperties processes the SAM_CreateComplete message and calls the SalOutlineItemGetProperty function to get the values of "MICROHELP" and "WNDTYPE" properties. It sets dfMicroHelp, rbMDI, and rbForm accordingly.

After the developer makes changes, and presses the OK push button, the dialog box sets the "MICROHELP" and "WNDTYPE" properties by calling the SalOutlineItemSetProperty function. Note that when the instance is created, the

property name does not exist for this instance, and this function creates a new property name.

On the other hand, if the developer chooses the Cancel push button, the dialog box does not change any properties of the instance and simply closes the dialog box.

```
Application Description: QCKPROP.APP
  Creating Your Own QuickObjects
  Setting Properties of cMicroHelp QuickObject
  Chapter 8
  Power Programming with SQLWindows
  by Rajesh Lalwani.
  Copyright (c) 1994 by Gupta Corporation.
  All rights reserved.
  Variables
   Number: nItem
   Number: nOutline
  Application Actions
   On SAM_AppStartup
     ! The QuickObject framework is initialized from the command
     ! line parameters passed by the SQLWindows designtime when
     ! the property editor is selected via the customizer
     ! or when an an instance of a QuickObject is dropped (the
     ! appropriate option is set in the QuickObject editor).
     ! The command lines parameters are defined as follows:
     ! strArgArray[1]:  Name of Dialog or Window to create
     ! strArgArray[2]:  The application (outline) identifier
     ! strArgArray[3]:  The item identifier
     ! strArgArray[4]:  The window handle of the calling app's
     ! design window
     Set nItem = SalStrToNumber(strArgArray[3])
     Set nOutline = SalStrToNumber(strArgArray[2])
     ! Create the specified window with the owner/parent window
     ! provided
     Call SalModalDialog(strArgArray[1],
          SalNumberToWindowHandle(SalStrToNumber(strArgArray[4])))
     Call SalQuit()
 Dialog Box: dlgSetProperties
  Title: cMicroHelp QuickObject Properties
  Description: This dialog box collects the properties
    HWNDTYPE and MICROHELP for the cMicroHelp
    QuickObject.
```

```
Contents
 Group Box: Show Micro Help on Status Bar of
 Radio Button: rbForm
  Title: Form Window
 Radio Button: rbMDI
  Title: MDI Window
 Background Text: Micro Help
 Data Field: dfMicroHelp
  Data
   Maximum Data Length: Default
   Data Type: String
   Editable? Yes
 Line
 Pushbutton: pbOk
  Title: OK
  Message Actions
   On SAM_Click
    Call SalOutlineItemSetProperty(nOutline, nItem,
      "MICROHELP", dfMicroHelp, SalStrLength(dfMicroHelp) + 1)
    If rbForm
     Call SalOutlineItemSetProperty(nOutline, nItem,
       "WNDTYPE", "Form", SalStrLength("Form") + 1)
    Else
     Call SalOutlineItemSetProperty(nOutline, nItem,
       "WNDTYPE", "MDI", SalStrLength("MDI") + 1)
    Call SalEndDialog( hWndForm, TRUE )
 Pushbutton: pbCancel
  Title: Cancel
  Message Actions
   On SAM_Click
    Call SalEndDialog( hWndForm, FALSE )
 Background Text:
  Don't forget to enable status bar on the form or MDI window.
 Frame
Window Variables
 String: strWndType
Message Actions
 On SAM_CreateComplete
  ! Maybe the developer has already defined properties
  ! Get them so that he or she can modify them.
  Call SalOutlineItemGetProperty(nOutline, nItem,
    "MICROHELP", dfMicroHelp)
  Call SalOutlineItemGetProperty(nOutline, nItem,
    "WNDTYPE", strWndType)
  If strWndType = "MDI"
```

```
    Set rbMDI = TRUE
  Else
    Set rbForm = TRUE
```

Listing 8.2 QCKPROP.APP—the custom interface application for cMicroHelp.

Using the QuickObject Editor

The QuickObject editor lets you specify the application that provides the designtime user interface for your QuickObject class. See Figure 8.3. You can launch the QuickObject editor by choosing Tools, QuickObject Editor... from the SQLWindows menu.

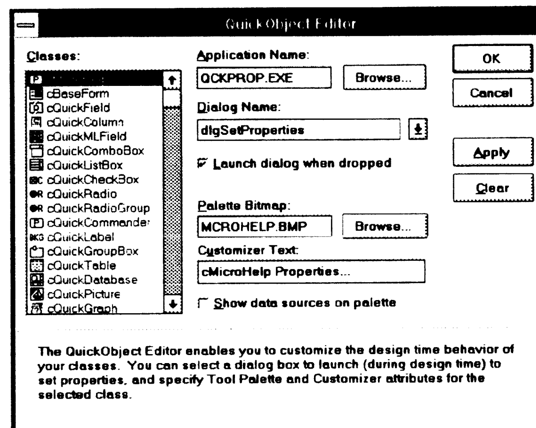

Figure 8.3 The QuickObject editor.

Application Name

Use Application Name to specify the name of the custom interface application. To debug your custom interfaces, you can specify an .APP file instead of an .EXE file as the custom interface application in the QuickObject editor. It is highly recommended that you make an .EXE from your custom interface and name the application for your class accordingly in the QuickObject editor before you distribute the class to developers.

Choose cMicroHelp in the Classes list box and specify QCKPROP.EXE.

Dialog Name

When you choose an application to associate with your class, SQLWindows displays a list of available form windows and dialog boxes. Choose one of these for the user interface for the QuickObject class.

For cMicroHelp, when you choose QCKPROP.EXE, the editor displays dlgSetProperties – the only window defined by this application.

A single SQLWindows application can contain multiple custom interfaces. You can develop whole libraries of QuickObjects whose custom designtime interfaces are contained totally in a single SQLWindows application.

Launch Dialog When Dropped?

In the QuickObject editor, you can specify whether you want your custom interface launched when a developer drops an object of the class. A developer using the class can always use your custom interface by launching it from the customizer (such as by choosing Quick Field..., Quick Radio..., etc.).

Palette Bitmap

QuickObjects can be displayed in the Tool Palette with bitmaps that visually represent their functions. In the QuickObject editor, you can specify the name of the file that contains the bitmap. Make these bitmaps 15 pixels wide and 14 pixels high. For cMicroHelp, choose MCROHELP.BMP which is on the companion disk.

Customizer Text

The QuickObject editor also lets you specify text (such as Quick Field...) to display as a menu item in the customizer. When a developer selects the menu item you specify, your custom interface application is launched. For cMicroHelp, specify cMicroHelp Properties... as the text.

Show Data Sources on Palette?

The QuickObject editor lets you specify whether data sources display or not in the Tool Palette.

Selecting this option lets you incorporate the standard QuickObject Visualizer Tool Palette behavior. When a developer chooses your class name, SQLWindows lists the available data sources and items for that source in the Tool Palette.

When a developer drops an object of the class, the "SOURCE" and "ITEM" properties are set to the values currently selected in the Tool Palette.

Using cMicroHelp QuickObject in an Application

MCRODEMO.APP defines a form windows as shown in Figure 8.4, where all the toolbar push buttons are instances of the cMicroHelp QuickObject. At this point, I encourage you to run this application and watch the status bar text change as you move the mouse over these push buttons.

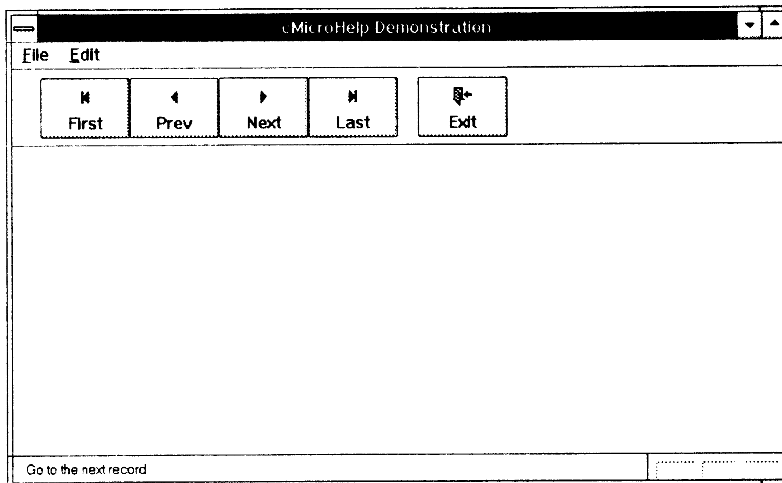

Figure 8.4 A form window with instances of cMicroHelp QuickObject.

Deriving New QuickObjects from Existing Ones

Now you know that QuickObjects are simply SQLWindows classes. Deriving new QuickObjects is similar to deriving new classes from one or more base classes.

QCKMODFY.APP derives a new QuickObject called cQuickFieldAutoEntry. This QuickObject is derived from both cQuickField (standard data field QuickObjects) and clsDfAutoEntry defined in AUTOENTR.APL (Chapter 4). See Listing 8.3.

An instance of cQuickFieldAutoEntry has properties of cQuickField. When a developer chooses the data field tool from the tool palette, and

cQuickFieldAutoEntry from the list of available classes, the bottom portion of the tool palette shows the available data sources and their columns. It also has the following property of clsDfAutoEntry: at runtime, when a user types the maximum number of characters (as defined in the customizer) in such a field, the focus automatically shifts to next control in the TAB order.

However, since both cQuickField and clsDfAutoEntry process SAM_AnyEdit and SAM_SetFocus messages, it is important that cQuickFieldAutoEntry uses the SalSendClassMessageNamed to make sure that *both* base classes process the messages.

SalSendClassMessageNamed

```
nMsgReturn =   SalSendClassMessageNamed ( tClassName, nMsg,
               nMywParam, nMylParam )
```

Invokes the message actions implemented or inherited by an object's class when called from either a derived class or from an object that is an instance of a class.

Use this function instead of SalSendClassMessage with a class that is the result of multiple inheritance. It lets you specify which base class message action you want to execute. tClassName must be a direct base class of the class from which this call is made.

tClassName is the ancestor's class name. nMsg is the message number. nMywParam is the wParam. nMylParam is the lParam.

nMsgReturn is the message return value. If the message has no return, nMsgReturn is zero (0).

```
Application Description: QCKMODFY.APP
  Creating Your Own QuickObjects
  Deriving a new QuickObject from cQuickField
  Chapter 8
  Power Programming with SQLWindows
  by Rajesh Lalwani.
  Copyright (c) 1994 by Gupta Corporation.
  All rights reserved.
   Data Field Class: cQuickFieldAutoEntry
    Description: This is a QuickField which is derived
      both from cQuickField and clsDfAutoEntry.
    Derived From
     Class: cQuickField
```

```
    Class: clsDfAutoEntry
   Message Actions
    On SAM_AnyEdit
     Call SalSendClassMessageNamed
        ( cQuickField, SAM_AnyEdit, wParam, lParam )
     Call SalSendClassMessageNamed
        ( clsDfAutoEntry, SAM_AnyEdit, wParam, lParam )
    On SAM_SetFocus
     Call SalSendClassMessageNamed
        ( cQuickField, SAM_SetFocus, wParam, lParam )
     Call SalSendClassMessageNamed
        ( clsDfAutoEntry, SAM_SetFocus, wParam, lParam )
```

Listing 8.3 Deriving cQuickFieldAutoEntry from cQuickField and clsDfAutoEntry.

9

Advanced Topics

In this chapter I introduce the following advanced topics:

- Pictures and OLE (Object Linking and Embedding).
- Drag and Drop.
- Visual Basic Extension (VBX) Controls.
- Team Programming.

While these topics deserve a more detailed discussion, they are beyond the scope of this book. However, in this chapter I attempt to present a brief overview of the concepts and give some examples where appropriate.

Pictures and OLE (Object Linking and Embedding)

In a SQLWindows application, you can use a picture object to contain the following:

- A graphic image.

  There are several places where you use a graphic image: corporate logo, images retrieved from a database or a file such as pictures of employees or pictures of the products, and images that fill the background of a form to enhance its visual appeal, etc.

- An OLE object.

  If you want to retrieve ASCII text from a database, you can display it in, say, a multi-line field. What do you do if you want to store, for example, a complete, formatted Microsoft Word document in a database and *remember* that it's a Word document upon retrieval? How about other objects such as a bitmap image that can be edited using Paintbrush, a Microsoft Excel

spreadsheet, or even a video clip? You can use a picture object in a SQLWindows application for such *objects*.

- A DOS file.

  A picture file can also contain any DOS files, such as SCRAPBK.APP or AUTHOR.TXT.

  You can insert any type of file into a picture, however only some files are associated with an application. For example, .BMP files are associated with Microsoft Paintbrush. When the user double-clicks on a .BMP package, Microsoft Paintbrush is invoked to edit the .BMP file.

  You can define associations between file extensions and programs in the [Extensions] section of WIN.INI.

  You can store a DOS file in a picture either by calling the SalOLEInsertFile function or by dragging a file from the File Manager and dropping it on an editable picture.

Graphic Images

A picture in a SQLWindows application can contain and display a graphic image of the following types:

Extension	Description
*.BMP, *.DIB	Device-Independent Bitmap
*.TIF	Tag Image File Format (TIFF)
*.PCX	Paintbrush
*.GIF	Graphics Interchange Format
*.WMF	Windows MetaFile
*.ICO	Icon File

At designtime, you can specify the contents of a picture by using the customizer. You can specify a file name by choosing Picture Contents, File Name... from the customizer.

File Storage

You can specify File Storage to be External or Internal. In case of external file storage, SQLWindows reads the picture image from a disk file at runtime and designtime. You should use external file storage if the picture files are constantly changing and you want to display the most current version of a picture. On the other hand, in case of internal file storage, SQLWindows stores the picture image in the application itself. You should use internal file storage if you are planning to distribute your application as an .EXE file and do not want to distribute the picture files separately.

Picture Transparent Color

If you specify a Picture Transparent Color, the background color of the picture replaces the specified color wherever it appears in an image. This applies to bitmaps only (.BMP files).

Picture Fit

You can specify one of the following at designtime or at runtime using the SalPicSetFit function:

Fit	SalPicSetFit Constant	Description
Scale	PIC_FitScale	Scales the image by a specified percentage. This is the default. The default values of the width and height scales are 100%.
Size to Fit	PIC_FitSizeToFit	Stretches or shrinks the image to fit in the picture.
Size for Best Fit	PIC_FitBestFit	Sizes the image to fit either the width or height of the picture.

If you are using Scale at designtime or PIC_FitScale, you can specify the scales for the width and height.

Tile to Parent

If you specify this customizer attribute as yes, the picture fills the background of the parent object. Also, the picture resizes itself with the resizing of the parent object.

Picture Functions

The following functions manipulate the contents of a picture. Some of them can also be used when the picture contains an OLE object.

Function Name	Description
SalPicClear	Clears the contents of a picture.
SalPicGetDescription	Retrieves a description of a picture's contents.
SalPicGetString	Copies the contents of a picture to a string.
SalPicSetFile	Sets the contents of a picture from a file.
SalPicSetFit	Sets the fit for a picture.
SalPicSetString	Sets the contents of a picture from a string.

About OLE

OLE stands for Object Linking and Embedding. An application that uses linked and embedded objects can contain many kinds of data in many different formats. With OLE, you can create a SQLWindows application that contains data from other applications. OLE gives the data the ability to "remember" the application that created it, and to transparently invoke that application for editing or "playing" the data. The end user can edit or play the data from inside the SQLWindows application, there is no need to switch between applications.

OLE Objects

OLE treats data as objects. An OLE object is any data that can be contained in another application and manipulated by a user; such as a Microsoft Word document, a bitmap image, or sound. An object can be a complete document or part of one. For example, a single cell, a range of cells, or an entire spreadsheet can be an object.

When an object is incorporated into another application (client), it maintains an association with the application that produced it (server). That association can be a link, or the object can be embedded.

Linking

If the object is linked, the client application provides only minimal storage for the data to which the object is linked, and the object can be updated automatically whenever the data in the original application changes. For example, if a range of spreadsheet cells were linked to a picture in a SQLWindows application, the data is stored in some other file and only a link to the data would be saved as the content of the picture of the SQLWindows application.

When you edit a lined object, you edit the original data. Any changes you make to this object appear in all other applications that are also linked to this data.

You can create links to data of *saved* documents only. For example, if you open Paintbrush and create a drawing, you must save it before you can create a link in a picture of a SQLWindows application to this drawing.

Embedding

If an object is embedded, all the data associated with it is saved as part of the file in which it is embedded. If a range of spreadsheet cells were embedded in a picture in a SQLWindows application, the data in the cells is saved as contents of the picture of the SQLWindows application, including any necessary formulas along with the name of the server for the spreadsheet cells. A user could double-click on this embedded object and the spreadsheet application would be started automatically for editing those cells.

When you edit an embedded object, the source document is not affected. For example, if you change an embedded drawing in a SQLWindows application, the drawing in the source file is not affected. Also, if you change the source file using Paintbrush, the embedded object in the SQLWindows application is not affected.

You should use embedding instead of linking when there is either no guarantee that the source document is always available (for example, when distributing an application) or when you want to store the actual data such as a Microsoft Word document in a database.

The presentation and the behavior of the data is the same for a linked and an embedded object.

Client and Server Applications

Applications that create and edit objects are called servers whereas the applications where you place objects are called clients. For example, if a Microsoft Word document is placed in a SQLWindows application, Microsoft Word is the server application, SQLWindows is the client application, and the Microsoft Word document is the object.

Some applications, such as Microsoft Word for Windows (version 2.0 onwards) and Quest can be both a server and a client. Another set of applications such as SQLWindows can only be clients. Finally, applications such as Paintbrush and Sound Recorder can only be servers.

OLE Verbs

The types of actions a user can perform on an object are called verbs. Two typical verbs for an object are Play and Edit.

Building an OLE Application—ScrapBook

With this background, let me now introduce ScrapBook. ScrapBook is a SQLWindows application which is used to organize a variety of objects such as Microsoft Word documents, Sound, bitmaps, etc. in a database. Using SCRAPBK.APP, you can retrieve the objects stored in the database, perform actions on them such as playing a sound or editing a Word document, insert new objects into the database, and update or delete existing ones. To remember what each object contains, you can use a Notes field to describe the contents of the object in plain English.

You can use ScrapBook to store frequently used objects such as:

- A Word document containing the fax cover sheet to be printed when you have used the last available copy.
- An Excel spreadsheet containing your stock portfolio. You can update the stock prices periodically and see the total value of your portfolio.
- Bitmaps for commonly used toolbar icons such as the ones used on Next, Last, Exit, etc. push buttons of Figure 9.1.

Figure 9.1 shows what the application ScrapBook looks like.

Structure of the SCRAPBOOK Table

The application SCRAPBK.APP stores the notes and the OLE objects in a table called SCRAPBOOK. The companion disk contains SCRAPBK.SQL which can be

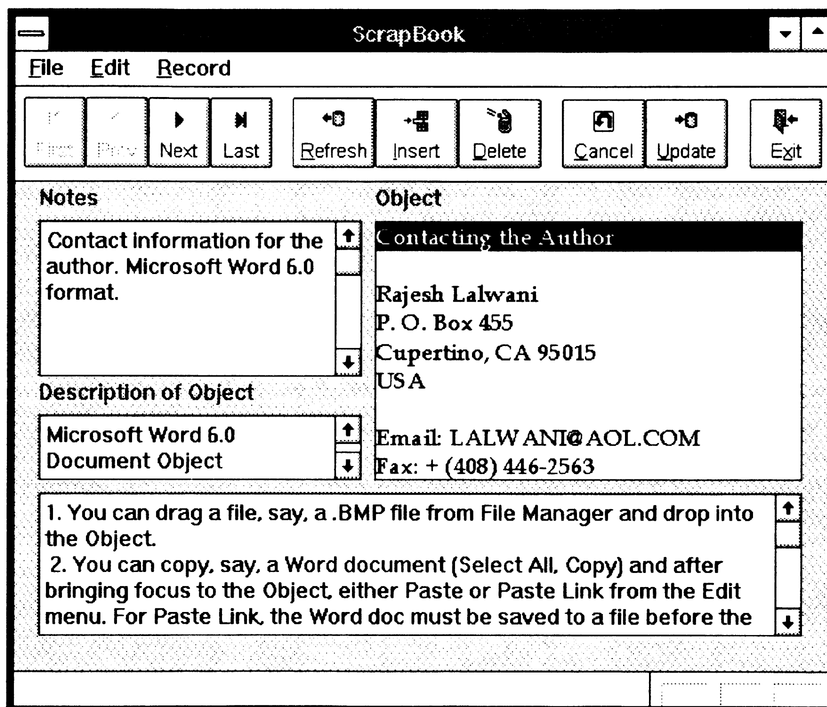

Figure 9.1 ScrapBook to manage variety of OLE objects.

used to create the table in a database. You can use SQLTalk (or WinTalk) for this. Start SQLTalk and follow these steps:

- Connect to a database where you want to create the SCRAPBOOK table.

- Use the LOAD command to load the SCRAPBK.SQL file. This creates the SCRAPBOOK table and inserts a few records in the table.
- Disconnect from the database and exit from SQLTalk.

Listing 9.1 shows a typical session.

```
connect gupta SYSADM/SYSADM;
load sql a:\scrapbk.sql;
disconnect gupta;
exit;
```

Listing 9.1 SQLTalk commands to create a SCRAPBOOK table.

You should have no problem in loading the SCRAPBK.SQL file. However, if you do, you can create a table called SCRAPBOOK with the following columns: NOTES of data type VARCHAR (254) and OBJECT of data type LONG VARCHAR.

Note that the data in SCRAPBK.SQL contains a Microsoft Word 6.0 object. If you do not have Word 6.0 on your system, you may choose to delete this record.

Form Window—frmMain

Listing 9.2 gives the overview of the main form window frmMain. Basically, the design of SCRAPBK.APP is very similar to DATABASE.APP of Chapter 3 and PHAD.APP of Chapter 5. Like DATABASE.APP, it has only one form window and its toolbar contains the push buttons for browsing through records, inserting a new record, deleting a record, etc. as seen in Figure 9.1. Internally, however, the form window is more like the form window of PHAD.APP which does the actual work in response to messages like PM_New, PM_GoToFirst, PM_Delete, etc. The push buttons on the toolbar simply post appropriate messages to the form window.

You can see that the picture picObject is derived from clsPicResetDirty and is editable. The form window menu contains two named menus menuFile and menuEdit and defines a new popup menu Record. menuFile contains menu items for Exit and About. The popup menu Record is similar to the one in PHAD.APP.

Notice the use of SAM_DropFiles for picObject. I explain this message handler as well as the basic concepts of Drag and Drop in the next section.

```
Application Description:
  SCRAPBK.APP
  OLE and Drag-and-Drop
  Chapter 9
  Power Programming with SQLWindows
  by Rajesh Lalwani.
  Copyright (c) 1994 by Gupta Corporation.
  All rights reserved.
 Form Window: frmMain
  Title: ScrapBook
  Description:
    This form window displays information about
    one record. It can also be used to modify
    this record, delete this record or
    insert a new record.
  Menu
   Named Menu: menuFile
   Named Menu: menuEdit
   Popup Menu: &Record
  Toolbar
   Contents
    Pushbutton: pbFirst
    Pushbutton: pbPrev
    Pushbutton: pbNext
    Pushbutton: pbLast
    Pushbutton: pbRefresh
    Pushbutton: pbExit
    Pushbutton: pbUpdate
    Pushbutton: pbNew
    Pushbutton: pbDelete
    Pushbutton: pbUndo
  Contents
   Background Text: Notes
   Multiline Field: mlNotes
    Class: clsMlResetDirty
    Data
     Maximum Data Length: 254
     String Type: String
    Display Settings
     Word Wrap? Yes
   Background Text: Object
   Picture: picObject
    Class: clsPicResetDirty
    Editable? Yes
```

```
  Message Actions
   On SAM_DropFiles
    If SalDropFilesQueryFiles
      ( hWndItem, strFilesBeingDropped ) > 1
     Call SalMessageBeep( 0 )
     Call SalMessageBox( 'You can only drop one file.',
       'Drag and Drop Error', MB_Ok | MB_IconExclamation )
     ! Executing a Return indicates no processing.
     Return FALSE
   On WM_RBUTTONDOWN
    ! The user has pressed right mouse button on the picture.
      Display the named menu menuEdit after bringing the focus
      to the picture.
    Call SalSetFocus( hWndItem )
    Call SalTrackPopupMenu( hWndForm, 'menuEdit',
      TPM_CursorX | TPM_CursorY, 0, 0 )
 Background Text: Description of Object
 Multiline Field: mlDescObject
  Data
   Maximum Data Length: 1024
   String Type: String
   Editable? No
  Display Settings
   Word Wrap? Yes
  Message Actions
   On PM_Initialize
    Call SalClearField( hWndItem )
 Multiline Field: mlInstructions
  Data
   Maximum Data Length: Default
   String Type: String
   Editable? No
  Display Settings
   Word Wrap? Yes
  Message Actions
   On SAM_Create
      '1. You can drag a file, say, a .BMP file from File
       Manager and drop into the Object.
       2. You can copy, say, a Word document (Select All, Copy)
       and after bringing focus to the Object, either Paste or
       Paste Link from the Edit menu. For Paste Link, the Word
       doc must be saved to a file before the copy operation.
       3. You can insert a new object by setting focus to the
       object and choosing Insert Object... from the Edit menu.
       4. You can click right mouse button on the Object to
```

```
        bring the Edit menu.'
Functions
 Function: OkToLoseChangesIfAny
 Function: BeginOperation
 Function: EndOperation
Window Variables
 ! When bFormDirty is TRUE, some fields have
   changed on the form
 Boolean: bFormDirty
 ! Remember where we are in the result set
 Number: nRowNumber
 ! When bInsert is TRUE it's insert operation
   not update
 Boolean: bInsert
 : hSqlFormSelect
  Class: clsSqlHandleSelectPhAd
 : hSqlFormUpdate
  Class: clsSqlHandlePhAd
 : hSqlFormInsert
  Class: clsSqlHandlePhAd
 : hSqlFormDelete
  Class: clsSqlHandlePhAd
 String: strSelectForm
 String: strUpdateForm
 String: strInsertForm
 String: strDeleteForm
 ! A long string to hold the object LONG VARCHAR
 Long String: strObject
 ! string to receive the files being dropped from
   the file manager.
 String: strFilesBeingDropped[*]
 Boolean: bOk
 Boolean: bRollback
 String: strUserPrompt
 String: currentRowID
 Boolean: bReturn   ! boolean return
 Number: nReturn   ! number return
```

Listing 9.2 Overview of the main form window frmMain.

Resetting Contents of a Picture

When a user wants to insert a new record, the form window sends a PM_Initialize message to all its child objects including the picture picObject.

picObject is derived from clsPicResetDirty which responds to PM_Initialize by calling SalPicClear.

SalPicClear

```
bOk = SalPicClear ( hWndToClear )
```

hWndToClear is the window handle (or name) of a picture.

bOk is TRUE if the function succeeds and FALSE if hWndToClear is not a valid picture.

Unfortunately, a SAM_AnyEdit message is not sent to a picture. So, to keep track of whether the contents of the picture picObject have changed, I make use of the field edit flag as you will see later during the discussion of the function OkToLoseChangesIfAny.

```
Picture Class: clsPicResetDirty
 Description: responds to PM_Initialize message.
 Message Actions
  On PM_Initialize
   ! Delete the contents of the picture field.
   Call SalPicClear( hWndItem )
```

Listing 9.3 Resetting contents of a picture.

Paste—Embedding an OLE Object

To embed an OLE object, a user must first cut or copy the desired object into the clipboard. After setting the focus to the picture, the user can choose Edit, Paste from the menu to embed the object in the picture.

SalEditCanPaste

```
bOk = SalEditCanPaste ( )
```

This function returns TRUE if there is data on the Clipboard that can be pasted into the editable picture with the focus. You can use this function in the Enabled When section for a Paste menu item.

bOk is TRUE if there is data on the Clipboard to be pasted and FALSE otherwise.

SalEditPaste

```
bOk = SalEditPaste ( )
```

Pastes data from the Clipboard into the data field, multiline field, table window column, or picture with the focus.

For a picture, if the application that cut or copied the data supports OLE, this function embeds the object. Otherwise, this function simply inserts a copy of the data without providing access to the source application.

bOk is TRUE if the function succeeds and FALSE if it fails.

```
Menu Item: &Paste
 Status Text: Inserts the Clipboard contents
 Keyboard Accelerator: Shift+Ins
 Menu Settings
  Enabled when: SalEditCanPaste()
 Menu Actions
  Call SalEditPaste()
```

Listing 9.4 Embedding an OLE Object.

Paste Link—Pasting an OLE Link

To be able to create a link, the user must first save the document before the copy operation. A user can choose Edit, Paste Link to create a link to the object in the clipboard.

The object is displayed in the application, but the data that defines that object is stored elsewhere. Any changes you make to this object are reflected in all other documents which are also linked to the object.

SalEditCanPasteLink

`bOk = SalEditCanPasteLink ( )`

This function returns TRUE if data from the Clipboard can be linked to the editable picture with the focus. You can use this function in the Enabled When section for a Paste Link menu item.

bOk is TRUE if an editable picture has the focus, and FALSE otherwise.

SalEditPasteLink

`bOk = SalEditPasteLink ( )`

Links the contents of the Clipboard with the picture that has the focus.

bOk is TRUE if the function succeeds and FALSE if it fails.

```
Menu Item: Paste &Link
  Status Text: Links the Clipboard contents to the field.
  Menu Settings
   Enabled when: SalEditCanPasteLink(  )
  Menu Actions
   Call SalEditPasteLink(  )
```

Listing 9.5 Pasting an OLE link.

Managing OLE Links

The menu item Links... of Edit menu is enabled only when the picObject contains a *linked* OLE object. When the user chooses Edit, Links... from the menu, it displays a dialog box as shown in Figure 9.2. Using this dialog box, the user can:

- Select manual or automatic updating. When a user chooses Automatic updating, the object is updated automatically whenever the server application modifies it.
- Cancel a link. When the user presses the Cancel Link push button, it breaks the link between the object and its source. The linked object remains in the picture as it was when the link was last updated but from this point on the object is no longer updated when there is a change in the source document. The source document is not affected by this operation.
- Change a link. If a user changes the name of the source file that contains a linked object, the link to that object is broken. Using Change Link... from this dialog box, the user can change the name of the file to which the object is linked so that the link works properly.
- Activate the linked object. When a user presses the Activate push button from this dialog box, it "plays" the object, such as playing a sound or running a movie.
- Edit the object. Using Edit push button, a user can start the server application (such as Microsoft Word or Recorder).

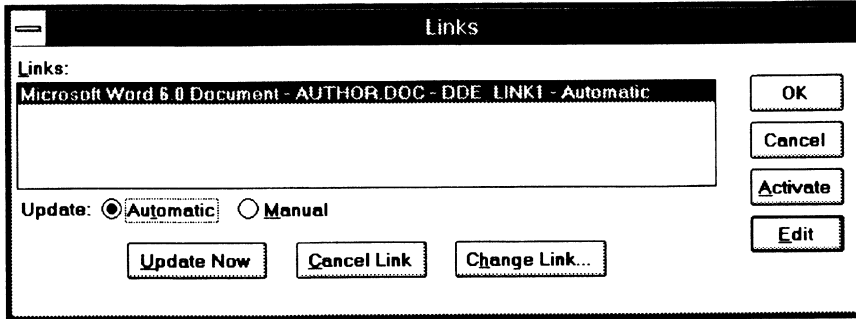

Figure 9.2 Dialog box to manage OLE links.

SalOLEAnyLinked

```
bOk = SalOLEAnyLinked ( )
```

This function returns TRUE if an application has any pictures that contain linked objects.

bOk is TRUE if the application has any pictures that contain linked objects and FALSE otherwise. You can use this function in the Enabled When section for a Links... menu item.

SalOLELinkProperties

```
bOk = SalOLELinkProperties ( hWnd )
```

Displays the Link Properties dialog box so that the user can manage the OLE links in an application.

hWnd is the window handle (or name) of the parent of the Link Properties dialog box.

bOk is TRUE if the function succeeds and FALSE if it fails.

```
Menu Item: &Links...
 Status Text: Manage the OLE link.
 Menu Settings
  Enabled when: SalOLEAnyLinked(  )
 Menu Actions
  Call SalOLELinkProperties( picObject )
```

Listing 9.6 Managing OLE links.

Performing Object Related Actions (OLE Verbs)

You can choose a named menu called Object Menu from the outline options. When the user chooses this at runtime, this context sensitive menu displays the menu items for the verbs supported by the OLE object which has the focus.

```
Object Menu
  Status Text: Perform Object related actions (verbs)
```

Listing 9.7 Performing object related actions (OLE verbs).

Inserting a New OLE Object

A user can choose Edit, Insert Object... from the menu which displays a dialog box that lists the OLE applications on the computer. The user can select an application, create a new object, and then embed the object in the picture.

Figure 9.3 shows the Insert New Object dialog box. The Object Type list box gathers the information from the Windows registration database REG.DAT which can be edited using REGEDIT.EXE.

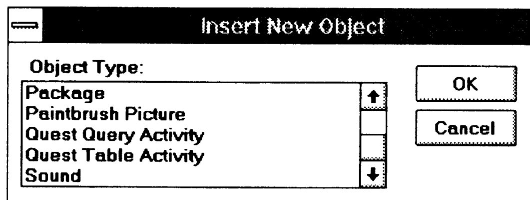

Figure 9.3 Dialog box displayed by Edit, Insert Object...

SalEditCanInsertObject

```
bOk = SalEditCanInsertObject ( )
```

This function returns TRUE if an editable picture has the focus. You can use this function in the Enabled When section for an Insert Object menu item.

bOk is TRUE if an editable picture has the focus and FALSE otherwise.

SalEditInsertObject

```
bOk = SalEditInsertObject ( )
```

Displays the Insert New Object dialog box where the user can specify the type of OLE object to insert. It also opens the selected OLE server application where the user can create the new object.

bOk is TRUE if the function succeeds and FALSE if it fails.

```
Menu Item: &Insert Object...
 Status Text: Insert a new Object
 Menu Settings
  Enabled when: SalEditCanInsertObject(  )
 Menu Actions
  Call SalEditInsertObject(  )
```

Listing 9.8 Inserting a new OLE object.

Creating a Popup Menu at Runtime

As you can see from Listing 9.2, the picture picObject processes the WM_RBUTTONDOWN (0x0204) message and creates the Edit popup menu at the current location of the mouse. Before creating the popup menu, it also sets the focus to the picture. This is a good example of creating a friendly user interface.

SalTrackPopupMenu

```
bOk =        SalTrackPopupMenu ( hWndProcMsgs, strMenuName,
             nFlags, nX, nY )
```

This function creates pop-up menus at runtime.

hWndProcMsgs is the window handle (or name) of the top-level window that processes messages generated by the pop-up menu.

strMenuName, a string, is the name of a pop-up menu accessible to hWndProcMsgs. For example, a menu defined by hWndProcMsgs or its MDI wirdow parent, or a global menu.

nFlags specifies how the pop-up menu displays. You can combine the following flag values using the OR (|) operator:

Constant	Description
TPM_CenterAlign	Center-align the pop-up menu.

Constant	Description
TPM_CursorX	Use the mouse cursor location instead of nX.
TPM_CursorY	Use the mouse cursor location instead of nY.
TPM_LeftAlign	Left-align the pop-up menu (default) to its horizontal screen coordinate. This coordinate identifies the cursor location, or the point specified by nX/nY.
TPM_LeftButton	Establish the left mouse button as the one with which to click on menu items (default).
TPM_RightAlign	Right-align the pop-up menu to its horizontal screen coordinate. This coordinate identifies the cursor location, or the point specified by nX/nY.
TPM_RightButton	Establish the right mouse button as the one with which to click on menu items.

nX is the position of the pop-up menu on the X axis. The value of this parameter is ignored if you set the TPM_CursorX flag in nFlags.

nY is the position of the pop-up menu on the Y axis. The value of this parameter is ignored if you set the TPM_CursorY flag in nFlags.

The nX and nY Parameters together represent a point (the top left, top middle, or top right) of the pop-up menu, depending on the nFlags value.

bOk is TRUE if the function succeeds and FALSE if it fails.

Detecting Changes to an OLE Object

There are two steps to detecting any changes to an OLE object. One to see if the user is still in the process of editing the object using the server application. This can be detected by calling SalOLEAnyActive function. If there are none, the second step is to find out whether the Field Edit flag is set or not.

SalOLEAnyActive

```
bOk = SalOLEAnyActive ( hWnd )
```

Indicates whether any OLE server applications are open for a given window.

hWnd is the window handle of a picture, a form window, a dialog box, or hWndNULL.

If hWnd is a picture, SalOLEAnyActive checks for an active server application for the picture contents.

If hWnd is a form window or dialog box, SalOLEAnyActive checks for an active server application for any picture on the form window or dialog box.

If hWnd is equal to hWndNULL, SalOLEAnyActive checks for an active server application for any picture in the application.

bOk is TRUE if an editor is open and FALSE otherwise.

Field Edit Flag

The field edit flag is set whenever the user changes the value of a data field, multiline text field, or table window column. For a picture, the field edit flag is set whenever an OLE server application is opened for editing. The field edit flag is not set if you make the change with a Set statement or a fetch from the database.

SalQueryEditFlag

```
bSet = SalQueryFieldEdit ( hWndField )
```

Returns the setting of the Field Edit Flag for a data field, multiline field, combo box, table window's context row cell, or picture.

The field edit flag is set whenever the user changes the value of a data field, multiline text field, or table window column. For a picture, the field edit flag is set whenever an OLE server application is opened for editing. The field edit flag is not set if you make the change with a Set statement or a fetch from the database.

This function does not clear the field edit flag of hWndField.

hWndField is the window handle (or name) of a data field, multiline field, combo box table window column, or picture.

bSet is TRUE if hWndField's field edit flag is set and FALSE otherwise.

SalSetEditField

`bOk = SalSetFieldEdit ( hWndField, bSet )`

Sets or clears the field edit flag for an editable data field, combo box, multiline text field, table window column, or picture.

hWndField is the window handle (or name) of a data field, multiline text field, table window column, or picture.

bSet is a boolean. If bSet is TRUE, the field edit flag is set. Otherwise, the field edit flag is cleared.

bOk is TRUE if the function succeeds and FALSE if it fails.

```
Function: OkToLoseChangesIfAny
 Description: See if the form is dirty (changes made
   to any child objects). If yes, ask user
   if it's ok to lose changes.
 Returns
  Boolean:
 Actions
  ! See if the server application corresponding to the
    object in picObject is currently active.
  If SalOLEAnyActive( picObject )
   Call SalMessageBeep( 0 )
   Call SalMessageBox(
     'Please close the object server application first.',
     'OLE Error', MB_Ok|MB_IconExclamation )
   Return FALSE
  If (bFormDirty or SalQueryFieldEdit( picObject )) and
    SalMessageBox( 'Lose Changes?', 'Confirmation',
    MB_YesNo | MB_IconQuestion | MB_DefButton2) = IDNO
   Return FALSE
  Else
   Set bFormDirty = FALSE
   ! Set the edit flag of picObject to FALSE.
   Call SalSetFieldEdit( picObject, FALSE )
   Return TRUE
```

Listing 9.9 Detecting changes to an OLE object.

Fetching an OLE Object from a Database

Fetching an OLE object from a database is slightly different. You cannot use a picture as an INTO variable in a SELECT statement to read the OLE object

directly into the picture. First, you must read the OLE object (LONG VARCHAR) into a SQLWindows Long String variable and then call the SalPicSetString function to copy the long string to the picture. You can also call SalPicGetDescription function to get the description of the OLE object contained in the picture.

In this application, I use the following SELECT statement:

```
'SELECT NOTES, OBJECT, ROWID   INTO :mlNotes, :strObject,
:currentRowID FROM SCRAPBOOK'
```

In SQLWindows, you use a Long String data type to transfer data to and from a LONG VARCHAR column of a database table. Because the length of such columns is unknown, special handling is needed within the SQLWindows interface to deal with them. In all other respects, a Long String data type is same as the String data type.

SalPicSetString

```
bOk = SalPicSetString ( hWnd, nFormat, strPicture )
```

Inserts the contents of a string into a picture.

hWnd is the window handle (or name) of a picture.

nFormat specifies the format of the picture contents:

Constant	Description
PIC_FormatBitmap	The picture contains a bitmap.
PIC_FormatIcon	The picture contains an icon.
PIC_FormatObject	The picture contains an OLE object.

strPicture, a string, is the memory image of a picture file. You can use SalFileRead to read a bitmap file into a string. You can also SELECT a LONG VARCHAR database column that contains a bitmap or icon file into a string.

bOk is TRUE if the function succeeds and FALSE if it fails.

SalPicGetDescription

```
nLength =    SalPicGetDescription ( hWndPict, strDesc,
             nMaxLen )
```

Retrieves a description of a picture's contents. An example of a description is "Microsoft Draw".

hWndPict is the window handle (or name) of a picture.

strDesc is a receive string. After a successful call it contains the description of the contents of hWndPict. For a graphic image, SalPicGetDescription returns one of the following values:

Contents of the Picture	strDesc
Device-Independent Bitmap	Gupta:BMP
Graphics Interchange Format	Gupta:GIF
Icon File	Gupta:ICO
Paintbrush	Gupta:PCX
Tag Image File Format	Gupta:TIFF
Windows MetaFile	Gupta:WMF
File	Gupta:DOSFILE
OLE object	Description of the object class that the server application recorded in the Windows registration database REG.DAT, followed by the word "Object".

nMaxLength is the maximum length of the text to retrieve.

nLength is the actual length (in bytes) of strDesc.

```
On PM_GoToFirst
 If OkToLoseChangesIfAny(  )
  ! Fetch the first row
```

```
    Set bOk = hSqlFormSelect.First(nReturn)
    If bOk
     If nReturn = FETCH_Ok
      ! The object was just read into :strObject. Place
        it in the picture field.
      Call SalPicSetString( picObject, PIC_FormatObject,
        strObject )
      Call SalPicGetDescription( picObject,
        mlDescObject, SalGetMaxDataLength( mlDescObject ) )
     ! We are not in the middle of insert
     Set bInsert = FALSE
     ! Remember where we are in the result set
     Set nRowNumber = 0
     ! Enable Next and Last. Disable First and Previous
     Call SalEnableWindow( pbNext )
     Call SalEnableWindow( pbLast )
     Call SalDisableWindow( pbFirst )
     Call SalDisableWindow( pbPrev )
```

Listing 9.10 Fetching an OLE object from a database.

Inserting or Updating an OLE Object in a Database

While inserting an OLE object in a database or updating an existing one, the process is just the reverse of what is done at the time of fetching an OLE object from the database. First you must copy the contents of the picture (OLE object) to a long string and then execute an INSERT or an UPDATE statement. To copy the contents of a picture to a long string, I call the SalPicGetString function. In this application I have used the following INSERT and UPDATE statements:

```
'INSERT INTO SCRAPBOOK (NOTES, OBJECT)
VALUES (:mlNotes, :strObject)'

'UPDATE SCRAPBOOK SET NOTES = :mlNotes, OBJECT = :strObject
WHERE ROWID = :currentRowID'
```

SalPicGetString

```
nLength = SalPicGetString ( hWndPict, nFormat, strPict )
```

Copies the contents of a picture to a string.

hWndPict is the window handle (or name) of a picture.

nFormat is the format of the picture contents: PIC_FormatBitmap, PIC_FormatIcon, PIC_FormatObject.

strPict is a receive string. After a successful call, it contains the contents of hWndPict.

nLength is the length (in bytes) of strPict. If the format is not available, nLength is equal to zero (0).

Compressing a String before Storing in a Database

This does not apply to SCRAPBK.APP, but you can copy the contents of a picture to a string and compress this string before storing it in a database. You can do so by calling the SalStrCompress function. If you do so, you must call SalStrUncompress after fetching the OLE object from the database before you copy it to the the picture.

SalStrCompress

```
bOk = SalStrCompress ( strString )
```

Compresses the specified string. Use this function to compress strings for storage on disk or in the database. Use this function for long strings, or when storing images and so on.

strString is a receive string to compress.

bOk is TRUE if the function succeeds and FALSE if it fails.

SalStrUncompress

```
bOk = SalStrUncompress ( strString )
```

Decompresses the specified string. Use this function to decompress a string that you compressed with SalStrCompress.

strString is a receive string to decompress.

bOk is TRUE if the function succeeds and FALSE if it fails.

```
On PM_Update
  ! See if we are in the middle of insert operation
  If bInsert
   ! Insert operation
   Call BeginOperation( 'Inserting the record...' )
   ! First read the object into strObject.
```

```
    Call SalPicGetString( picObject, PIC_FormatObject,
      strObject )
    If hSqlFormInsert.Prepare() and hSqlFormInsert.Execute()
      ! Time to commit the transaction.
      Call BeginOperation( 'Committing...' )
      Call hSqlFormInsert.Commit()
      ! Mark the form as not dirty (no changes yet)
      Set bFormDirty = FALSE
      ! Mark the edit field of picObject as FALSE
      Call SalSetFieldEdit( picObject, FALSE )
      ! We are not in the middle of the insert
      Set bInsert = FALSE
      ! Now re-fetch the current row of the result set.
        In case, the UPDATE failed last time and
        rollback occurred, don't fetch the row as
        the SELECT may not have survived the
        rollback.
      If not bRollback
        Call hSqlFormSelect.FetchRow(nRowNumber, nReturn)
  Else
    ! Normal Update - not insert operation
    Call BeginOperation( 'Updating the record...' )
    ! First read the object into strObject.
    Call SalPicGetString( picObject, PIC_FormatObject,
      strObject )
    If not hSqlFormUpdate.Prepare()
      Call SalMessageBox(
        'Could not prepare the Update statement',
        'Serious Error', MB_Ok | MB_IconStop )
      ! Time to quit
      Call SalSendMsg( hWndForm, SAM_Close, 0, 0 )
    If hSqlFormUpdate.Execute()
      ! UPDATE was successful, now COMMIT it.
      Call BeginOperation( 'Committing...' )
      Call hSqlFormUpdate.Commit()
      ! Mark the form as not dirty (no changes yet)
      Set bFormDirty = FALSE
      ! Mark the edit field of picObject as FALSE
      Call SalSetFieldEdit( picObject, FALSE )
      ! Now re-fetch the same row of the result set.
        One reason we have been keeping track of
        nRowNumber. This refetching is necessary to get
        the new ROWID so that if the user makes a change
        to this record again without first refreshing,
        invalid ROWID error would not occur.
```

```
    In case, the UPDATE failed last time and
    rollback occurred, don't fetch the row as
    the SELECT may not have survived the
    rollback.
  If not bRollback
   Call hSqlFormSelect.FetchRow(nRowNumber, nReturn)
Call EndOperation(  )
```

Listing 9.11 Inserting or updating an OLE object in a database.

Drag and Drop

Using SQLWindows you can write applications where the user can use the mouse to drag from one window and drop into another. You can use drag and drop features in your application to provide a user friendly interface. Let me give you some examples where you can use these features:

- You can allow the user to drag a file from the File Manager and drop it into a picture. As you can see from the instructions on the form window of ScrapBook, SCRAPBK.APP provides this feature.
- You can allow the user to drag an item of a list box or a row of a table window and drop it in a Delete push button. You can write code so that such an action deletes the corresponding record from the database.
- From a list of reports available, you can let the user drag one or more to a picture of a printer causing those reports to be printed.
- You can let the user fill a data field by dragging another data field or an item from a list box to this data field.

Dropping Files from the File Manager

As you know, ScrapBook lets the user drag a file from the File Manager and drop in the picture so that the picture contains a package object containing that file. An editable picture automatically allows files to be dropped on it. If you want to allow other window object, such as a data field, to have files dropped on it, you can call SalDropFilesAcceptFiles.

SalDropFilesAcceptFiles

```
bOk = SalDropFilesAcceptFiles ( hWnd, bAccept )
```

Indicates whether a window can accept a file from the File Manager.

hWnd is the window handle (or name) of a window.

bAccept is a boolean which specifies whether hWnd can accept a file from the File Manager. The default for editable pictures is TRUE; the default for all other window types is FALSE.

bOk is TRUE if the function succeeds and FALSE if it fails.

SAM_DropFiles

Listing 9.12 reproduces a section from Listing 9.2 where the picture picObject responds to the SAM_DropFiles message.

```
On SAM_DropFiles
  If SalDropFilesQueryFiles
    ( hWndItem, strFilesBeingDropped ) > 1
   Call SalMessageBeep( 0 )
   Call SalMessageBox( 'You can only drop one file.',
     'Drag and Drop Error', MB_Ok | MB_IconExclamation )
   ! Executing a Return indicates no processing.
   Return FALSE
```

Listing 9.12 Picture picObject responds to the SAM_DropFiles message.

This message is sent to a column, data field, multiline field, list box, combo box, picture, and custom control. SQLWindows only sends this message to windows that have enabled file dropping. It is sent when the user drops a file or files from File Manager on the object.

You can call the SalDropFilesQueryFiles function to get the names of the files dropped on the object. SalDropFilesQueryFiles also returns the number of files that were dropped or 0 if the function fails. You can only call the SalDropFilesQueryFiles during SAM_DropFiles message processing.

You can call the SalDropFilesQueryPoint function to get the location of the mouse in the window where the user dropped the file or files.

By default, file dropping is enabled for editable picture objects. To avoid this default processing, execute a Return statement in the SAM_DropFiles message processing for a picture object and do not perform any other processing.

When a user drops on a picture, SQLWindows inserts an OLE Package Object (the same as an object created by PACKAGER.EXE) in the picture that represents the file. For example, if the user drops a *.BMP file on a picture, SQLWindows inserts an OLE Package Object that launches Microsoft Paintbrush. To associate an application with a file, SQLWindows reads the [Extensions] section of WIN.INI to find the application for that file extension.

SalDropFilesQueryFiles

```
nFiles = SalDropFilesQueryFiles ( hWndTarget, sArrayFiles )
```

Retrieves the names of the files dropped on a window. Receipt of the SAM_DropFiles message indicates that the user dropped files onto the window.

hWndTarget is the window handle (or name) of the window on which the files were dropped.

sArrayFiles is a receive string array. After a successful call to the function this array contains the names of the dropped files.

nFiles is the number of dropped files if the function succeeds and zero (0) if the function fails. SalDropFilesQueryFiles fails unless it is called as the result of receiving the SAM_DropFiles message.

Drag and Drop between Application Windows

The following windows can be source windows in a SQLWindows application:

- Data Field
- Multi-line Field
- List Box
- Combo Box
- Picture
- Table Window

Any SQLWindow object can be a target window except objects that do not have a Message Actions section (such as a Frame) to process any SAM_Drag* messages.

SQLWindows can start the drag process automatically if a window returns TRUE from the SAM_DragCanAutoStart message. Auto dragging begins when the user holds down the left mouse button over the window and does not move the mouse for a short period of time.

A SQLWindows application can also start the drag process at any mouse-down event such as SAM_Click or WM_RBUTTONDOWN (0x0204) by calling SalDragDropStart function.

From this point on, SQLWindows informs both the source and target windows about the drag mode events by sending messages.

Source Window Messages

The following table lists the messages that are sent to a source window:

Message Sent to a Source Window	Description
SAM_ DragCanAutoStart	SAM_DragCanAutoStart is sent to a top-level window, data field, multiline field, list box, combo box, and picture. SQLWindows sends this message to ask a window whether it wants auto dragging. Return TRUE to enable auto dragging. If you do not process this message or you return FALSE, then SQLWindows does not enable auto dragging.
SAM_DragStart	Indicates that drag mode has started.
SAM_DragEnd	Indicates that drag mode has ended.
SAM_DragNotify	Indicates that a mouse action occurred while in drag mode. The lParam contains one of the following constants: SAM_DragEnter, SAM_DragMove, SAM_DragExit, and SAM_DragDrop.

Target Window Messages

The following table lists the messages that are sent to a target window:

Message Sent to a Target Window	Description
SAM_DragEnter	Indicates that the mouse has moved into the window while in drag mode.
SAM_DragMove	Indicates that the mouse has moved onto the window while in drag mode.

Message Sent to a Target Window	Description
SAM_DragExit	Indicates that the mouse has moved off of the window while in drag mode.
SAM_DragDrop	Sent when the user drops the mouse on a target window.

Drag and Drop Related Functions

The following table lists drag and drop related functions along with their brief description:

Function Name	Description
SalDragDropDisableDrop	Disables dropping while in drag mode.
SalDragDropEnableDrop	Enables dropping while in drag mode.
SalDragDropGetSource	Retrieves the handle of the source window and the location of the mouse when the user initiated drag mode.
SalDragDropGetTarget	Retrieves the handle of the target window and the location of the mouse.
SalDragDropStart	Initiates drag mode.
SalDragDropStop	Ends drag mode.
SalDropFilesAcceptFiles	Specifies whether a window can accept a file from the File Manager.
SalDropFilesQueryFiles	Retrieves the names of the files dropped on a window.

VBX Controls

Just like object oriented programming, VBX controls also make the software development process very similar to assembling various pre-fabricated parts.

Many custom controls have been developed for the Microsoft Visual Basic environment. You can purchase needed VBX controls along with documentation from their respective vendors. Beginning with version 4.1, you can use these VBX controls in SQLWindows applications.

Examples of VBX controls include a graph VBX control, a spin VBX control, and even a mini-spreadsheet VBX control. SQLWindows comes with a SWSPIN.VBX control in the SAMPLES directory.

VBX is implemented as a Dynamic Link Library (DLL)—with a different extension of .VBX. VBX stands for Visual Basic Extension. A VBX is generated using Visual C++ and the Microsoft Control Development Kit (CDK).

A VBX developer must follow a set of guidelines that are decribed in the documentation and online help of Microsoft Visual Basic CDK. The VBX must be compatible with Microsoft Foundation Classes (MFC).

Placing a VBX Control

Just like any other child object, you can place a VBX control on a form window, dialog box or a toolbar by selecting the custom control tool from the tool palette and dropping it on the parent window. At this point, you are asked to select the VBX file that contains the desired VBX control. See Figure 9.4.

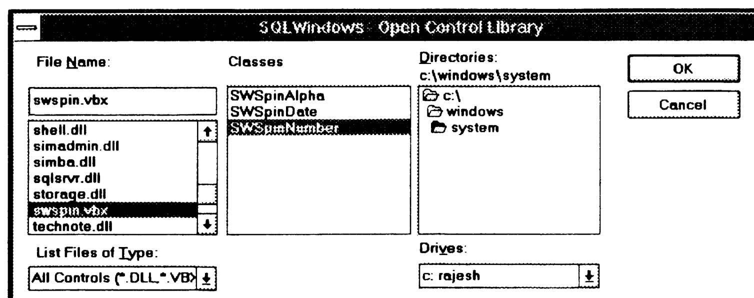

Figure 9.4 Selecting a VBX file and Microsoft Windows class name.

Unless the VBX provider advises otherwise, you must place the .VBX and other accompanying files in the WINDOWS\SYSTEM directory.

VBX Interface

The interface to a VBX control consists of *events* and *properties*. Each VBX control has its own set of properties and events. You can get the documentation for the control from the provider of the VBX.

Events

An event is like a programmer message and represents the actions that the VBX control recognizes. For example, the SWSPIN.VBX control recognizes VBX_SpinUp and VBX_SpinDown events as shown in Figure 9.5.

There are standard events that most VBX controls recognize. A VBX can also have custom events that are unique to it. SQLWindows automatically maps events to messages (such as VBX_SpinUp). You can respond to these message in the Message Actions section just like you respond to, say, SAM_Click for a push button.

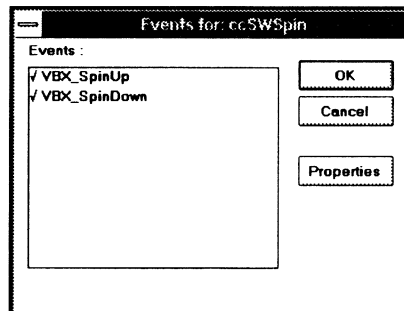

Figure 9.5 Events recognized by ccSWSpin.

Properties

Properties define the attributes of an object, such as size, color, font, etc. Figure 9.6 shows a partial list of the properties that can be defined for SWSPIN.VBX.

There are standard properties that most VBX controls have. A VBX can also have custom properties that are unique to it.

At designtime, you can change the properties by choosing Properties... from the customizer. There are SAL functions to change and retrieve the properties at runtime.

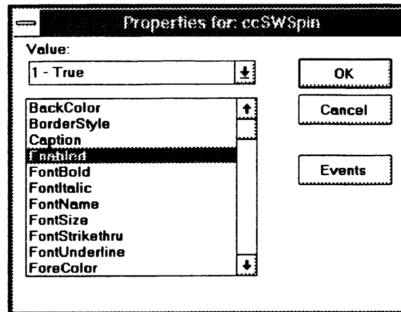

Figure 9.6 Partial list of properties for ccSWSpin.

VBX-Related SAL Functions

The following table provides a list of VBX-related SAL functions along with their brief description:

Function	Description
SalVBXAddItem	Adds an item to a container control such as a VBX list box.
SalVBXGetError	Returns an error number that indicates why a SalVBX* function failed.
SalVBXCloseChannel	Channels are file numbers that some VBX controls use to access files in Visual Basic. This function disassociates a file from a channel.

Function	Description
SalVBXGetNumParam	Returns a numeric parameter for a VBX event. Some VBX events have numeric parameters that lParam points to. Retrieve the parameters with SalVBXGetNumParam. When a VBX event occurs, lParam contains a pointer to the event parameter list.
SalVBXRefresh	After you change one or more visual VBX properties, call SalVBXRefresh to repaint the control so that it reflects the changes you made.
SalVBXRemoveItem	This function removes an item from a container control.
SalVBXSetPicture	Lets more than one VBX control share a picture. Memory for the same picture is not allocated multiple times, rather a reference count for the picture is incremented when it is copied.
SalVBXSetProp	Call this function to set a property in a VBX control. You use SalVBXSetProp to set both string properties and numeric properties. SQLWindows looks up the property data type and converts strPropValue to the appropriate data type.

Listing 9.13 shows a simple example to demonstrate the use of SWSPIN.VBX control. Notice that VBX_SpinDown and VBX_SpinUp have been defined as System constants by SQLWindows.

```
Application Description:
   VBX.APP
   Demonstrates the use of SWSPIN.VBX control.
   Make sure SWSPIN.VBX is in WINDOWS\SYSTEM directory.
```

```
Global Declarations
 Constants
  System
   Number: VBX_SpinDown = 1025
   Number: VBX_SpinUp = 1024
Form Window: frmMain
 Title: SWSPIN.VBX Control
 Contents
  Custom Control: ccSWSpin
   Display Settings
    DLL Name: swspin.vbx
    MS Windows Class Name: SWSpinNumber
   Message Actions
    On SAM_Create
     Set nValue = 1
     Call SetValue (hWndItem, nValue)
    On VBX_SpinUp
     Set nValue = nValue + 1
     Call SetValue (hWndItem, nValue)
    On VBX_SpinDown
     Set nValue = nValue - 1
     Call SetValue (hWndItem, nValue)
 Functions
  Function: SetValue
   Description: This function sets the value of
     the property 'Caption' of hWndVBX
   Parameters
    Window Handle: hWndVBX
    Number: nValue
   Local variables
    String: strValue
   Actions
    ! First convert the number to string
    Call SalNumberToStr( nValue, 0, strValue )
    ! Now set the Caption property
    Call SalVBXSetProp( hWndVBX, 'Caption', strValue, 0 )
 Window Variables
  Number: nValue
```

Listing 9.13 A simple example to demonstrate the use of SWSPIN.VBX control.

Team Programming

SQLWindows' unique TeamWindows component offers the team coordination and management capabilities required for collaborative programming.

If you are doing application development in a team environment where several programmers work on a *project*, you may consider using TeamWindows. TeamWindows manages the individual pieces of code, called *modules*. A module can be anything—a SQLWindows application (.APP file), SQLWindows library (.APL file), a bitmap (.BMP file), or even a Microsoft Word for Windows document. TeamWindows provides a repository for source code and version control. The check-in/check-out facility enables team members to share and modify modules.

TeamWindows maintains a central data dictionary of database structural information, including table and column names, primary and foreign-key relationships.

Figure 9.7 gives an overview of TeamWindows components. Let me explain some of the basic concepts and terms.

TeamWindows

Project Development Manager is the most visible and most used of all the TeamWindows components. Therefore, it is normally referred to as TeamWindows. It contains tools for managing SQLWindows projects, generating applications, maintaining template libraries, and creating screens.

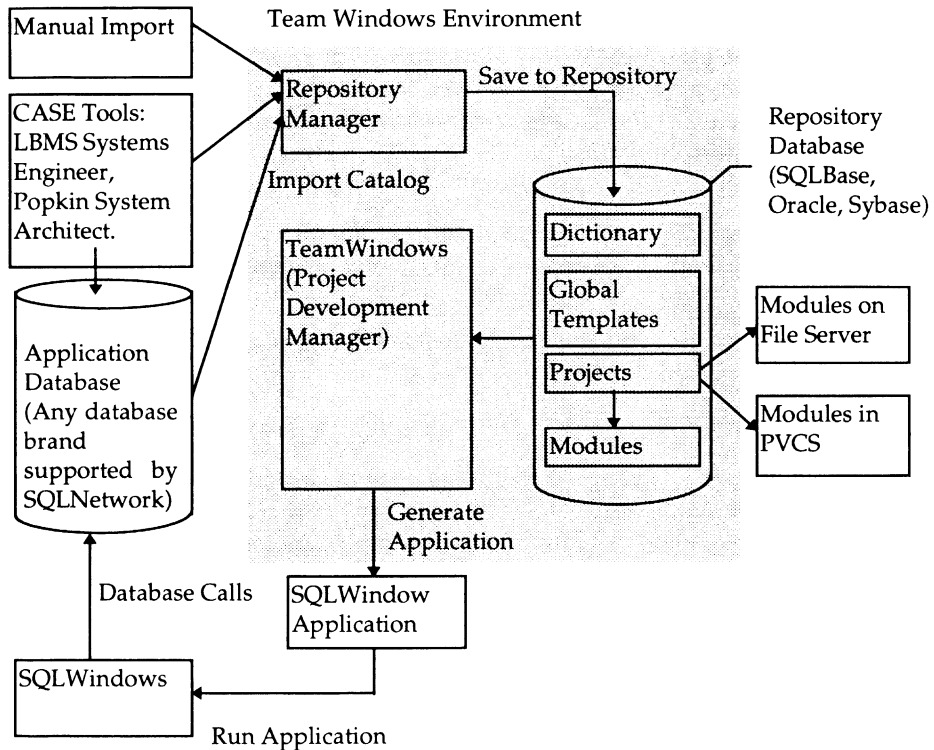

Figure 9.7 Overview of TeamWindows Components.

Repository

The repository is a centralized, multi-user database that contains a data dictionary of information about your application database. This information about the application database can be imported either manually, automatically or by using a CASE tool such as LBMS Systems Engineer or Popkin System Architect.

The repository also holds project-related information and files, including a current copy of each project module and information about the contents of the screens within each SQLWindows application.

Note that the repository database is different from the application database. The application database refers to the database used by the final application generated using TeamWindows. It can be an inventory management database, or a finance database—it depends on the purpose of the application you are developing.

Even though the application database can be any database brand supported by SQLNetwork, the repository database can be only of the following brands:

- SQLBase
- Oracle
- Sybase

Project

A project in TeamWindows relates to your real world development project. It can be as small as a single SQLWindows application or it may contain hundreds of SQLWindows applications and related files. A project is the highest level entity in TeamWindows. A project consists of:

- One or more SQLWindows application modules.
- Possibly, one or more non-SQLWindows modules, such as template libraries, bitmaps, documents, or reports.
- An associated application database.
- Development standards.

- A project manager and other staff.

You can either create a new project from scratch, or import from a .PRJ file which was created earlier by exporting an entire project from TeamWindows.

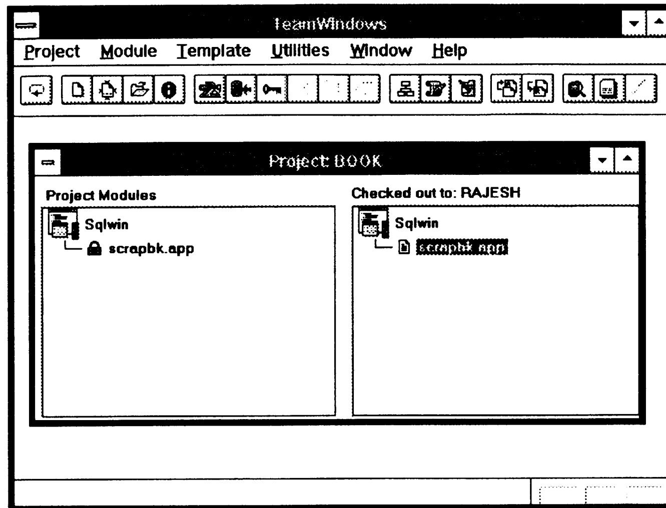

Figure 9.8 TeamWindows—a user RAJESH is currently working on a project BOOK. Module SCRAPBK.APP has been checked out to RAJESH.

TeamWindows Users

In addition to two special users, TWADM (TeamWindows administrator) and SYSADM (SQLBase system administrator), initially TWADM and later a project administrator can create and manage other TeamWindows users. Each TeamWindows user belongs to one of the following four access levels:

- **Project Administrator**

  Project administrators have the highest access level. Project administrators have the authority to access and maintain all projects in the TeamWindows environment. They are the only users who can create a new project or manage users.

- **Project Manager**

  A project manager is assigned to a project by a project administrator at the time of creation of the project. A project manager can create new modules and assign staff to the project among other things.

- **Developer**

  Developers work on the modules of the project(s) that they belong to. Developers can check out modules and work on them before they check them back in.

- **Tester**

  Testers are responsible for ensuring the quality of the modules. They have restricted access to the modules.

Other privileges that can be selectively granted to TeamWindows users are as follows:

- Template Engineer
- Database Administrator
- Test Sign Off
- Log All Errors

Modules

Every TeamWindows project contains one or more modules. Each module represents a unit of work that needs to be completed during the project's development phase.

Module Types

A module can be any type of file that you need to associate with your project. For example, it may be a SQLWindows application file (.APP file) or a non-SQLWindows file such as a bitmap (.BMP file), a document (.DOC file), or a report query (.QRP file).

TeamWindows defines the following module types by default:

Module Type	File Extension	Description
Bitmap	.BMP	Standard windows bitmap file.
ERWin	.ER1	Erwin data model.
Icon	.ICO	Standard windows icon file.
Library	.APL	SQLWindows library.
Report	.QRP	SQLWindows report template.
Sound	.WAV	Standard windows waveform file.
Sqlwin	.APP	SQLWindows application.
Template	.TL	TeamWindows template library.
Text	.TXT	Standard windows text file.
WordDoc	.DOC	Microsoft Word for Windows document.
WriteDoc	.WRI	Windows Write document.

Defining New Module Types

You are not limited to working with modules of type defined by TeamWindows. You can choose Project, Module Types..., New... to define a new module type. For a new module type, you need to define the following:

- Type—a unique code to identify the module type. If it is a file that SQLWindows can edit, you should check the SQLWindows Application check box so that TeamWindows can differentiate it from other module types.
- Title.
- Icon File. TeamWindows uses the icon you specify to represent project modules within project folders on the TeamWindows main screen.
- File Extension. For example, enter BMP for bitmaps.

- Edit Command—filename and path for the editor to use in association with this module type. For example, enter 'PBRUSH.EXE' for bitmap files. Alternatively, you can leave the Edit Command field blank to use whatever editor is currently associated with this file extension in windows.
- Starter file—filename and path details for this module type's starter file.

Module Storage Methods

You can choose to store the source code for your project's modules in one of three different ways, either as LONG VARCHAR entries in the repository database itself, as special files on a network file server, or in PVCS.

The choice you make should be based on administrative considerations for your project. If you store your modules within the repository database you have better security over your source code and it is easier for you to take backups of your project work. If you store your modules as files in your TeamWindows Workspace path, they consume less disk space.

You can also specify whether you want a module to be compressed before storage. A compressed module takes less disk space but takes longer during a check-out or check-in operation.

Checking Out and Checking In a Module

A user who has been assigned to a project and who is at necessary access level, can check out a module from the repository. TeamWindows copies the latest version of the module in a directory specified by the user. TeamWindows locks this module so that no other user can check out this module.

Later, when the user is done with making changes to this module, the user can check in the same module. The module is copied to the repository and the module is unlocked so that other users working on this project can check it out.

If for some reason, the copy that was checked out to the user is deleted, or the user wishes to lose all the changes made to this copy, the module in the repository can be unlocked without the usual step of checking it in. Of course, only a user with appropriate access level can unlock a module.

Extracting a Module

There are several occasions when you only want to get the latest copy of a module from the repository without affecting other users working with this module. For example, you may want the latest copy of the toolbar icons so that you can place them on push buttons, or the latest copy of a library file (.APL file) which is being developed by another user but you need to include it in the application module (.APP file) that you are working on.

In such cases, instead of checking out a module, you can extract the module from the repository. TeamWindows places the latest copy of the module in your personal directory without placing any lock on this module in the repository. Other users (including yourself) are free to check out and work on this module.

Module Relationships

TeamWindows lets you define a relationship between a module (parent module) and other modules (child modules) that it (parent module) needs. Later, when you check out the parent module, TeamWindows gives you an option to extract all the modules that are related to the parent module. This way, it is very easy for you to get all the bitmaps, libraries, etc. in one simple step. Also, once the relationship is defined, you do not have to remember which other modules are needed by the module that you are checking out.

TeamWindows Development Levels

When a module is first created, it is at the Development level. When a developer checks out this module, works on it and checks it back in, the developer can optionally mark it ready for promotion. Depending on whether this module is shared, the module can be promoted to either Shared or Test level. After a tester has tested the module, the tester can promote the module to the Production level (provided the tester has the Test Sign Off privilege).

The promotion facility is optional and if you work on a small project, you may elect to keep all the modules at the Development level and manage testing and building executables etc. on your own.

Figure 9.9 shows the TeamWindow development levels.

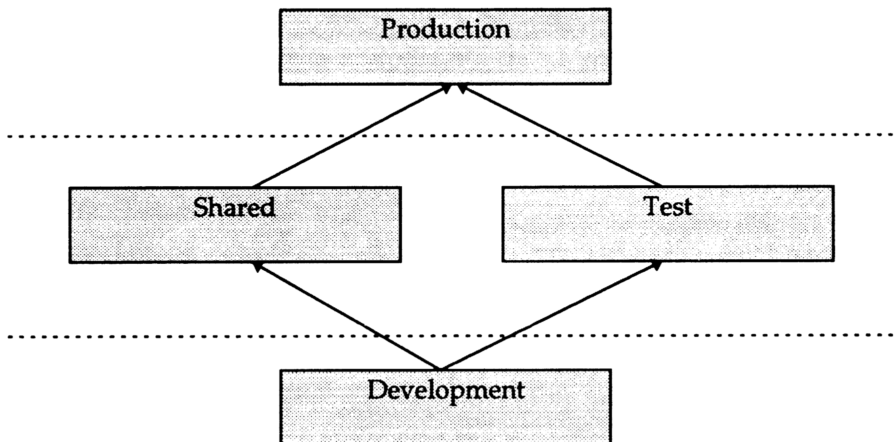

Figure 9.9 TeamWindows development levels. A module can be either at Shared or Test level depending on whether the module is Shared or not.

Templates

Templates are special screen definitions that provide a basis for application development in the TeamWindows environment. In effect, templates are a 'delivery mechanism' for predefined SQLWindows object classes (such as data field classes, form window classes, etc). The application developer is able to use

TeamWindows to pick and choose the classes supported by a particular template with confidence that the overall integrity of the screen being developed is controlled by the underlying template.

The idea behind templates is that while an application may consist of many different screens, it generally contains only a few different types, or classes, of screens. By providing a storage place for common screen classes and common screen functions, templates reduce the work necessary to design and build your screens.

When you use a template, you no longer have to start from scratch each time you develop a screen. Tasks such as adding required objects (like OK and Cancel buttons), coding common screen functions, and designing basic screen layouts have already been taken care of by the template designer.

Impact Analysis

While TeamWindows provides several useful tools and reports, one of the most important tool is the impact analysis. You can use the Impact Analysis screen to review the impact of database changes on projects, modules, and screens that use the database you select.

The Impact Analysis screen provides detailed information about each database change, including the name of all the projects, SQLWindows applications, screens, and window objects impacted as well as the nature of the change and, if relevant, the old and new values.

Glossary

abstract class: An abstract class has no instances and is created for the sole purpose of organizing a class hierarchy or defining methods and variables that apply to lower-level classes. See instance.

archive: An organized collection of data. Also, a place where data is kept.

argument: See parameter.

application: A SQLWindows program written for a user that applies to the user's work.

Application Actions: A section in the outline that contains procedural code that executes when the application receives messages.

application window: A form window, table window, or MDI window.

attribute: A distinct feature assigned to an object.

backend: See database server.

background text: A field that contains static text.

base class: The class from which a given class is derived. A derived class inherits its base class' data structure and behavior. Also called superclass, ancestor, and parent class. Contrast with derived class. See also class, inheritance and object-oriented programming.

behavior: The set of operations that an object can perform on its data.

bitmap: A series of bits where one or more bits correspond to each display pixel. For a monochrome bitmap, 1 bit corresponds to 1 pixel. In a gray-scale or color bitmap, multiple bits correspond to each pixel to represent shades of gray or color.

browse: A mode where a user queries a database without necessarily making additions or changes. In a browsing application, a user needs to examine data before deciding what to do with it. A browsing application lets the user scroll forward and backward through data.

buffer: A memory area that holds data during input/output operations.

CASE: The acronym for Computer Aided Software Engineering. CASE consists of software tools that automate or support the process of designing and programming software systems.

case sensitive: A condition in which data must be entered in a specific lowercase, uppercase, or mixed-case format.

cell: The intersection of a row and a column in a table window.

character: A letter, digit, or special character (such as punctuation mark) that represents data.

check box: A box that represents an option. When checked, the option is on; when not checked the option is off.

class: A template that specifies the data and behavior of an object. Objects are created at run time and are instances of a class, while classes are a static description of a set of objects. You create classes in hierarchies, and inheritance lets you pass the data and behavior in one class down the hierarchy. See also base class, derived class, inheritance, object, and object-oriented programming.

class function: A function that you write in a class definition. A class function is the implementation of an object's behavior. An object's functions are shared by all objects in the same class. Also called method or member function.

class variable: Stores data that is shared by all objects in a class. Contrast with instance variable.

client: A computer that accesses shared resources on other computers running as servers on the network. Also called front-end or requester. See also: server and database server.

collapse: Hiding lower outline levels.

column: In a table window, a complete vertical line of cells. See also table window and row.

In a database, a data value that describes one characteristic of an entity. A column is the smallest unit of data that can be referred to in a row. A column contains one unit of data in a row of a table. A column has a name and a data type. Sometimes called field or attribute. See also database and row.

Commander: A QuickObject used to manipulate data which is always linked to a data source. See Data Source and Visualizer.

combo box: A child object that contains a data field and a list box. The list box contains scrollable, pre-defined choices the user selects to fill a data field. See data field.

contents: The objects in form windows, table windows, dialog boxes, MDI windows, and toolbars.

cursor: A work space in memory that is used for processing a SQL command. This work space contains the return code, number of rows, error position, number of select list items, number of program variables, rollback flag, and the command result. A cursor is part of a Sql Handle.

Cursor is also a name for a mouse pointer.

Customizer: A list of attributes used to define the behavior of a selected object. The customizer is invoked by dragging the window grabber over the object and double-clicking on it.

data field: A one-line data entry/output field.

Data Source: A QuickObject used to define the data access connection. A data source is used to define the data for specific windows in an application. See Commander and Visualizer.

data type: One of the standard forms of data that SQLWindows can store and manipulate.

database: A collection of interrelated or independent pieces of information stored together without unnecessary redundancy. Client applications can read and write a database.

database server: A DBMS that a user interacts with through a client application on a different computer. Also called backend. See also engine.

DBMS (database management system): A software system that manages the creation, organization, and modification of a database and access to data stored within it. A DBMS provides centralized control, data independence, and complex physical structures for efficient access, integrity, recovery, concurrency, and security.

derived class: A class defined from an existing class (its base class). A derived class inherits its base class' data structure and behavior and it can change (override) the inherited data structure and behavior. It can also add its own data structure and behavior. All of the derived class' data structure and behavior-changed, new, and inherited-is inherited by its own derived classes. Also called descendant, subclass, and child class. Contrast with base class. See also inheritance and object-oriented programming.

design window: A top-level window at designtime. You use the Window Editor to add child objects to a design window.

designtime: An environment in which you can create and change an application. Contrast with runtime.

dialog box: A single-function window that displays data and messages and accepts input. In general, a dialog box requests or provides information to the user. See also modal dialog box, modeless dialog box, and system modal dialog box.

diamond: The symbol that represents the beginning of an item in the outline.

disabled: A menu item or menu that cannot be chosen; the menu item or menu title appears dimmed or gray.

embed: To insert an object completely within a OLE client application. An embedded object contains a presentation format (bitmap or MetaFile), a data structure that identifies the server, and the native data provided by the server. A user can edit an embedded object directly in the client application. Editing an embedded object starts the server and sends the native data back to that server. See also link, object, and OLE.

encapsulation: This term has two related meanings:

- Combining data and procedures together in a class or object.
- Making the behavior of a class or object visible while hiding the details of its implementation.

Also called data hiding or information hiding. See also class and object-oriented programming.

engine: A DBMS that a user interacts with through a client application on the same computer. See database server.

expand: Displaying lower outline levels.

fetch: To retrieve one or more rows of data from a database.

File Handle: A data type that identifies an open file.

form window: A top-level window used for data entry and display.

format: The appearance of data. Currency, percentage, decimal, date, time, invisible, numbers, and unformatted are examples.

Fourth-generation language (4GL): A type of computer language that accepts system requirements as input and generates a program to meet those requirements as output. This type of language is used for clearly defined procedures such as the generation of menus, forms, and reports.

frontend: See client.

function: A routine that performs a task that needs to be done several times but at different places in an application. SQLWindows has 5 types of functions: built-in system functions, internal functions, window functions, external functions, and class functions.

global: A variable, function, or object known in all parts of an application. Constants are always global. See also local and scope.

grid: A pattern used to align objects in a design window.

group box: A box that labels a set of related child objects.

group separator: Separates two contiguous sets of radio buttons.

handle: A number that identifies a window, a database connection, or an open file. An application gets a handle by calling a Sal* or Sql* function. The application then uses the handle in other functions to refer to the window, database connection, or file. An application does not know the actual value of the handle.

hierarchy: The relationship between classes. See also base class, derived class, inheritance, and object-oriented programming.

hWndForm: A variable that contains the handle of the parent.

hWndItem: A variable that contains the handle of a child object.

icon: An image that represents a minimized application or window.

icon file: A file that contains a window icon, specified in .ico format.

impact analysis: A report that shows the impact of a global change to an application or group of applications. For example, the impact of a database change on every screen, in all modules in a project.

instance: An object that belongs to a particular class. For example, America is an instance of the class country.

instance variable: A variable that contains data that is private to a specific object in a class. Also called field, member, and slot. Contrast with class variable.

inheritance: Arranging classes in a hierarchy so that classes can share the data and behavior of other classes without duplicating the code. A derived class automatically includes the data and behavior of one or more base classes. Derived classes can add their own data and behavior and can redefine inherited data and behavior. A derived class can inherit from one base class (single inheritance) or more than one base class (multiple inheritance). See also base class, class, derived class, and object-oriented programming.

interface: The external view of an object which hides the code that implements its behavior.

library: A collection of SQLWindows objects such as form windows, dialog boxes, or class definitions that are shared by more than one application.

link: In OLE, to create a reference to an object in an OLE client application whose native data is stored in another file maintained by an OLE server for that object. The client application contains only a presentation format such as an icon or bitmap and a data structure that identifies the linked file. The user can edit a linked object directly in the client application. When the object changes in the server application, the changes appear (automatically or on demand) in the client application. See also embed, object, and OLE.

In programming, to connect programs compiled or assembled at separate times so they can be executed together.

list box: A read-only object that displays a list of items.

local: A variable, function, or object that is known only in a single part of an application. See also global and scope.

master/detail form: The master table contains key information. The detail table stores detailed information. The master table forms the base from which all details are extracted.

maximize: To expand a window so it fills the entire screen.

MDI (Multiple Document Interface): A user interface model created by Microsoft.

MDI window: A workspace used to manage the placement of form windows and top-level table windows.

MDI child window: An object created inside an MDI window, which is its parent window. The parent window owns the child window.

menu: A list of choices from which you can select an action. A menu appears when you click the menu title in the menu bar.

menu item: A choice in a menu or menu bar.

message: The way that objects interact with each other. An object (the sender) sends a message to another object (the receiver) to make a request or notify it of something. The receiver can ignore it or take some action.

Messages make it easy to reuse code. The internals of an object can change while the messages remain the same. Code changes are then highly localized.

Message Actions: A section in the outline where you code actions that are responses to messages.

minimize: To collapse a window into an icon.

modal dialog box: A dialog box that suspends the application until the user closes the dialog box.

modeless dialog box: A dialog box that does not stop processing within other windows.

module: A set of programs, broken down into components, commonly called modules. Each module has its own set of procedures and data. In general, modules are designed to be as independent of each other as possible.

mouse pointer: A graphic symbol that shows the location of the mouse on the screen. The mouse pointer is usually an arrow, but can change to other shapes during some tasks. Also called cursor.

multiuser: The ability of a database server to provide its services to more than one user at a time.

multiline field: A multiple line field for data entry and display.

multiple inheritance: A class or object that inherits properties from more than one class.

null: A value that means the absence of data. Null is not considered equivalent to zero or to blank. The value of null is not considered to be greater than, less than, or equivalent to another value, including a null value.

object: A window object is a visual element on the screen such as a table window, push button, or menu.

An OLE object is a piece of information such as a drawing, chart, or sound that is linked with or embedded in another application. See also embed, link, and OLE.

An OOP object is a representation of a software entity such as a user-defined window or a user-defined variable. An object has the data structure and behavior specified by its class (which is its blueprint). Also called instance, occurrence, or instantiation. See also class, encapsulation, and object-oriented programming.

object-oriented programming (OOP): Programming that uses objects which combine data and behavior. Objects use the data structure and behavior of their class. See also class, encapsulation, inheritance, object, and polymorphism.

OLE (Object Linking and Embedding): A method of sharing information between different Windows applications. By linking and embedding objects, you can combine different types of information in a single application. A SQLWindows application can act as an OLE client. See also link, embed, and object.

operator: A symbol or word that represents an operation to be performed on the values on either side of it. Examples of operators are: arithmetic (+, -, *, /), relation (=.!=, >, <, >=, <=), and logical (AND, OR, NOT).

outline: The statements that define the objects and procedural logic in an application.

Outline Options bar: The part of the Outline window that lists options that are appropriate for adding to the current outline section.

outline items: Each line of text in an outline.

Outline window: The window that displays the application outline. You use the Outline window to write and test an application with SQLWindows.

parameter: A value for a function that defines the data or controls how the function executes. Also called argument or operand.

polymorphism: The ability for different objects to respond to the same request in different ways. For example, both a push button and a scroll bar can respond to a paint message, but the actions they take are different. The sender does not need to know the type of object that is responding to the message. Also called overloading. See also OOP.

populate: To fill a table window with data from a source such as a database, array, or file

popup menu: See menu.

profile: A set of format specifications.

push button: An object that invokes an action when the user clicks it.

query: A request for information from a database, optionally based on specific conditions. For example, a request to list all customers whose balance is greater than $1000. You give queries with the SQL SELECT command.

QuickForms: An automatic or manual feature for building forms from Data Sources.

QuickObject: Pre-defined objects, defined in a class library, for use in SQLWindows applications. A QuickObject is a data source, a visualizer or a commander. See data source, visualizer, and commander.

QuestWindow: An object that lets the user build Quest applications from a query or table activity.

radio button: A circle that represents an option that can be on or off. In a group of radio buttons, only one can be on at a time.

Repository: A centralized, multi-user database that holds a data dictionary of information about the application database. It also holds project-related information, files, and miscellaneous information.

row: In a table window, a complete horizontal line of cells in a table window.

In a database, a set of related columns that describe a specific entity. For example, a row could contain a name, address, and telephone number. Sometimes called record or tuple. See also table and column.

runtime: The time during which a user executes a program.

SAL (SQLWindows Application Language): A procedural language for writing actions that execute at runtime.

scope: The part of an outline where the name of a variable, function, or object is known. See also local and global.

scroll bar: A rectangle with an arrow on each end and a square box. A window can have either a vertical or horizontal scroll bar, or both. A horizontal scroll bar runs along the bottom of the window, a vertical scroll bar runs along the right side of the window.

scroll box: The box in a scroll bar. The scroll box indicates the position relative to the entire window's contents. Also called thumb or elevator box.

section: A parent item and all its children.

server: A computer on a network that provides services to client applications. See also: client and database server. The term server is also used to refer to applications in DDE, ReportWindows, and OLE.

single inheritance: A mechanism where a class can make use of the methods and variables defined in the class above it on that branch of the class hierarchy.

single-user: A database server that can only provide its services to one user at a time.

standard class: A built-in SQLWindows object. See QuickObject.

status bar: A bar at the bottom of the designer window that shows the setting of the Num Lock, Scroll Lock, and Caps Lock keys, as well as other information such as the current menu pick or toolbar pick.

structured development: A top-down approach to program design in which a program is consistently broken down into components. Each of the components is decomposed into a sub-component. This type of development involves separate teams of programmers, who each work on various components, which are assembled into a complete program at the end of the implementation phase of development. Contrast object-oriented programming.

subclass: A class that is a special case of another class. For example, deer is a special case of Mammal. See derived class.

With the inheritance process, all the subclasses for a given class use the methods and variables of that class. Large groups of objects are programmed only once, in the higher-level class, and the subclass is used only to add or modify behavior as required for special case classes.

sizing pointer: A pointer you use to change the size of an object.

SQL (Structured Query Language): A standard set of commands used to manage information stored in a database. These commands let users retrieve, add, update, or delete data. There are four types of SQL commands: Data Definition Language (DDL), Data Manipulation Language (DML), Data Query Language (DQL), and Data Control Language (DCL). Pronounced ess-que-ell or sequel.

Sql Handle: A data type that identifies a connection to a database. See also cursor.

SQLWindows: A graphical SQL application development system for Microsoft Windows.

statement: A unit in the application outline that specifies an action for the computer to perform.

string: A sequence of characters treated as a unit.

system function: A built-in SQLWindows function that performs an often-needed task.

system modal dialog box: A dialog box that suspends all Window applications until the user closes the dialog box.

table: The basic data storage structure in a relational database. A table is a two-dimensional arrangement of columns and rows. Each row contains a like set of data items (columns). Also called relation.

table window: An object that displays data in a tabular format (columns and rows). A table window can be a top-level or child window.

template: The definition of an object created at run time.

toolbar: A rectangular area at the top of the window where objects are placed. The objects represent the functions of an application.

Tool palette: The moveable Tool palette contains icons that represent graphical objects like push buttons and data fields. These icons are used to add objects to an application during designtime. See designtime.

top-level window: A form window, table window (that is not a child), or dialog box.

user mode: A designtime mode in SQLWindows when you compile and run an application.

value: Data assigned to a constant or a variable.

variable: A named item that can be any of a given set of values. Contrast with constant.

Visualizer: A QuickObject that displays and interacts with the data from a data source. Examples of Visualizers are data fields, check boxes, radio buttons, and business graphics. See also Commander and Data Source.

window: A rectangular area on the screen where an application receives input from the mouse or keyboard and displays output.

Window Editor: SQLWindows' drawing functions. You use the Window Editor to add child objects to a design window.

window function: A function that you write in a top-level window (form window, dialog box, or table window) or in an MDI window. The scope of a window function is in actions in the window.

Window Grabber: A mouse pointer that moves an object.

Window Handle: A data type that identifies a single instance of a particular window.

Windows: A graphical user interface from Microsoft that runs under DOS. In Windows, commands are organized in lists called menus. Icons (small pictures) on the screen represent applications. A user selects a menu item or an icon by pointing to it with a mouse and clicking.

Applications run in windows that can be resized and relocated. A user can run two or more applications at the same time and can switch between them. A user can run multiple copies of the same application at the same time.

Index

—&—

&, 138
&&, 138

—.—

.APC, 32
.APL, **28**
.APP, 28
.BMP, 260
.DIB, 260
.EXE, 32
.GIF, 260
.ICO, 260
.PCX, 260
.QRP, 195
.TIF, 260
.WMF, 260

—A—

accessories, 112
Allbase/SQL, 2
allow row sizing, 161
Alt, 138
alternate background, 210
ampersand, 138
animate, **32**
animate, slow, **32**
AppendMenu, 142
Application Actions, 38
application libraries, **28**
application outline, 6, **27**
application outline editor, 26
applications, 79
architecture, 112
AS/400, 2
ASCII, 166
assembly language, 115
attributes, 6
auto entry, 80
AUTOENTR.APL, 80
average, 202

—B—

background text, 138
background text tool, 207
base class, 85, 107
batch printing, 236
bFormDirty, 115
binary search, 114, 176
bind variables, **61**
blank lines, 208
borders, 208
box tool, 208
break, 31
break group, **199**
broadcast, 137
browse, 37, 86
browsing, 113
browsing applications, 52
buffer, 82
business applications, 79

—C—

C, 115, 117
C++, 117
cache, **161**

calling convention, 117
cc:Mail, 10
cell type, **161**
check box, 138, 161, **237**
check in, 301
check out, 301
child table window, 99, 160
child window, 112, 120
Cincom Supra, 2
class, **80**, **81**, 93, 107, 108
class variable, **86**, 102
client, 52, 73, 163
client application, 264
clsDfAutoEntry, **80**, 81
clsDfCurrency, **85**
clsDfNumber, **84**, 85
clsdlgReport, 122
clsFrmBrowse, 86, 91, **97**, 102
clsFrmCorpBase, 107
clsSqlHandle, 86, 102, 122
clsSqlHandleSelect, 86, 102, 122
clsWndCorporate, **107**
cMicroHelp, 245
colors, 208
column, 99, 138
column break, 208
columns, 164, 233
combo box, **214**
commander, 13, 25
COMMIT, 51, 53, 189
communication, 112
compiler, 7, **32**
concurrency, 52, 53
connect, 111, 266
consistency, 53
context row, **172**
Corporate edition, 3
corporate standards, 107
count, 202
CountNull, 202
CountUnique, 202
cQuickCheckBox, 24
cQuickField, 23
cQuickRadioGroup, 23
cQuickTable, 21
cross tab, **224**
cross tabular report, **224**
CS, **51**
CURRENT OF, 52
cursor context, 51
Cursor Stability, 50, **51**
custom control, 6
custom controls, 117
custom interface application, 249
customizer, 6, **11**, 48, 80, 81, 82, 83, 101, 112, 163

—D—

data dictionary, 7, 295
data field, 138
data field class, 83
data items, 204
data source, 13, 18, 21
data type, 84, 85
database application, 37
Database Manager, 2
DATABASE.APP, **37**, 143
DB2, 2
DBP_PRESERVE, **51**
DDL, 51, 52
debugger, 7
default, 92
default binding, 91
default database name, 46
default error processing, 92
default isolation level, 53, 92
default password, 46
default user name, 46
delete, 37, 66, 73

deploying an application, 34
deployment, 3
design mode, **20**
design window, 5
designtime, 246
detail block, 199
developer, 299
development levels, 302
device drivers, 117
dialog box, 137
dialog box class, 122
discardable, **163**
disconnect, 111, 266
displaying a report, 211
distribution, **34**, 45, 117
dlgLogin, 43, 112
dlgSqlError, 128, 129
DLL, 6, **117**
DML, 52
drag and drop, **285**
drop down list, 161
DW_Default, **82**
dynamic binding, 91
dynamic link library, 6. **See DLL**
dynamic table window, 167

—E—

early binding, 91
EM_SETSEL, 81, 83
e-mail, 3, 117
embedding, 263
enabled when, **144**
EnableMenuItem, 115, 136, **142**
error text, 102, 106
ERROR.SQL, 47
events, VBX, 291
exclusive lock, 47, 51, 72
executable, **32**
execute, 60
EXEHDR.EXE, 118
Exit, 76
external data type, 118
external functions, **115**
extracting a module, 302

—F—

FAR PASCAL, 117
FETCH_Delete, **63**
FETCH_EOF, **63**, 93
FETCH_Ok, **62**
field tool, 206
file storage, 261
focus, 80, 111, 137, 138
focus row, **171**
form window, 53, 110, 137
form window class, 86
format, 85
formula, 201, **204**
formula editor, **204**
formula name, 205
FRMBROSE.APL, 129
FRMBROWS.APL, 104
frontend result sets, 50
functional class, 86
functions, 204

—G—

GDI.EXE, 117
general window class, **127**
GetProperty, 249
GetSystemMenu, 115, 136, **142**
graphic images, 260
group box, 132, 138
group separator, 132
GUEST table, 104
Gupta client/server suite, 1
Gupta Corporation, 1

GUPTA database, 37, 104

—H—

HP Allbase/SQL, 2
hWndForm, **99**
hWndItem, **45**
hWndNULL, **43**, 70

—I—

IBM AS/400, 2
IBM DB2, 2
impact analysis, 304
implementation details, 108
implicit commit, 53
Informix, 2
Ingres, 2
inheritance, **85**, 97, 98
input items, **199**
input message buffer, 52
input totals, **201**
insert, 37, 70, 73
InsertMenu, 142
instance, **81**, 113
instance variable, **82**, 85, 86, 93, 97, 102, 113, 141
internal data type, 118
INTO, **61**, 62, 166
invoice, 195
INVOICE.QRP, 197
isolation level, 53, 72
isolation levels, **51**
item identifier, 250

—K—

KERNEL.EXE, 117
key stroke, 82
keyboard accelerator, 46

—L—

landscape, 231
late binding, **91**, 106
launching a report, 217
LBMS Systems Engineer, 8
libraries, **28**
library, 6, 117
line tool, 208
lines per row, 160
linking, 263
LOAD, 111, 266
lock, 51
login, 37, 112
login dialog box, **38**
long string, **279**
LONG VARCHAR, 266, 279
loop, 70
Lotus cc:Mail, 10
Lotus Notes, 4, 13
lParam, 47

—M—

mailing labels, 111, **232**
maintenance of code, 107
MAPDLL.EXE, 118
maximize, 139
maximum, 202
MB_DefButton2, 60
MB_IconQuestion, 60
MB_YesNo, 60
MCROHELP.APL, 245
MDI, 17
MDI window, 53, **109**
menu item, 138
menuEdit, 121
menuMDIWindows, 121
menuOLEEdit, 121
message, 55

MF_BYCOMMAND, 143
MF_BYPOSITION, 136, 143
MF_DISABLED, 136, 143
MF_ENABLED, 143
MF_GRAYED, 136, 143
Microsoft Mail, 10
Microsoft SQL Server, 2
minimize, 139
minimum, 202
minimum height, 236
MIS, 107
mnemonics, **138**
modal dialog box, 43
modeless dialog box, 53
ModifyMenu, 142
module relationships, 302
module storage methods, 301
module types, 299
modules, 7, 295, 299
multiline field, 48, 114
Multiple Document Interface. See MDI window
multiple inheritance, **85**
multi-user environment, 37, 62
MyValue, **82**

—N—

named menus, **120**, 164
named properties, **246**
named transactions, **76**
network, 52
Network edition, 3
network traffic, 52
NEWAPP.APP, 121

—O—

object, **81**
object linking and embedding, 262
object-oriented programming, **79**
ODBC, 2
OLE, **259**, 262
OLE links, 272
OLE verbs, 264, 274
OOP, **79**
operators, 205
options bar, 7, **29**
options box, **29**
Oracle, 2
ordinal number, 117, 118
orientation, 231
OS/2 Database Manager, 2
outline, **27**, 138
outline identifier, 249
outline options, 7, **29**
outline views, **27**
override, 91, 92
owner, 53

—P—

page, 51
page break, 208
page footer, 198
page header, 198
performance, 10, 52, 91, 93, 203
PHAD table, 111
PHAD.APP, **109**
PIC_FitBestFit, 261
PIC_FitScale, 261
PIC_FitSizeToFit, 261
PIC_FormatBitmap, 61, **279**
PIC_FormatIcon, **279**
PIC_FormatObject, **279**
picture, **260**
picture fit, 261
picture functions, 262
picture tool, 207
picture transparent color, **101**, 261

Popkin System Architect, 8
popup edit, 161
popup menu, 121
prepare, 60
pre-process, **203**
primary key, 37
printer, 111
printing a report, 211
private, 93
privileges, 299
production, 45
program group, 33
program item, 33
Program Manager, 33
programmer messages, 119
project, 295, 297
project administrator, 298
project manager, 299
properties, 24, 246
properties, VBX, 291
protected, 93
public, 93
push button, 138
push button class, 113

—Q—

QCKPROP.APP, 250
qualification, 93
query, 159
Quest, 2, 195
Quest Reporter, 2
QuestWindow, 174
Quick Objects, 5
QUICK.APP, 17, 25
QuickField, 23
QuickForms, 17
QUICKFRM.APP, 15
QuickObject editor, **253**
QuickObjects, 3, 5, **12**, 20, **245**
QuickObjects, properties, 24
QuickTable, 21

—R—

radio buttons, 129, **132**, 138
Read Only, 52
Read Repeatability, **53**
reason, 102, 106
Release Locks, **52**, 53
remedy, 102, 106
report activity, 195
report footer, 198
report header, 198
report templates, 195
REPORT.APL, 116, 122
reports, 111, **195**
ReportWindows, 7, 195, 198
repository, 297
resource, 61
Resources, 39, **45**
restart event, **202**, 230
restore, 138
result set, 51
result set mode, 50
RETAIL_CUSTOMER, 15
RL, **52**, 53, 72
RO, **52**
ROLLBACK, 51, 53
row flags, 172
ROW_Edited, 163, **172**
ROW_Hidden, **172**
ROW_HideMarks, **172**
ROW_MarkDeleted, **172**, 186
ROW_New, 163, **172**
ROW_Selected, **172**
ROW_UnusedFlag1, **172**
ROW_UnusedFlag2, **172**
ROWID, 62, 66, 74, 181, 185, 189, 190
RPT_PrintAll, 218

RPT_PrintDraft, 218
RPT_PrintNoAbort, 218
RPT_PrintNoErrors, 219
RPT_PrintNoWarn, 219
RPT_PrintRange, 219
RR, **53**
running an application, 20
runtime files, **35**

—S—

SAL, 6
SalBringWindowToTop, **139**
SalClearField, 70
SalCreateWindow, 53, 128
SalDestroyWindow, 59
SalDisableWindowAndLabel, **238**
SalDragDropDisableDrop, 289
SalDragDropEnableDrop, 289
SalDragDropGetSource, 289
SalDragDropGetTarget, 289
SalDragDropStart, 289
SalDragDropStop, 289
SalDrawMenuBar, 145
SalDropFilesAcceptFiles, **285**, 289
SalDropFilesQueryFiles, **286**, 289
SalEditCanInsertObject, **274**
SalEditCanPaste, **270**
SalEditCanPasteLink, **271**
SalEditInsertObject, **274**
SalEditPaste, **270**
SalEditPasteLink, **271**
SalEnableWindowAndLabel, **238**
SalEndDialog, 46, 50, 59
SalGetFirstChild, 70
SalGetFocus, 134, **137**
SalGetMaxDataLength, 81
SalGetNextChild, 70
SalGetWindowState, **139**
SalGetWindowText, 134, **137**
SalIsNull, 217
SalIsWindowEnabled, 145
SalListPopulate, **214**, 216
SalLoadAppAndWait, 139
SalMessageBox, 83
SalModalDialog, **43**, 83
SalNumberToStrX, **97**
SalOLEAnyActive, **276**
SalOLEAnyLinked, **273**
SalOLELinkProperties, **273**
SalOutlineItemGetProperty, **252**
SalOutlineItemSetProperty, **250**
SalPicClear, **270**
SalPicGetDescription, **280**
SalPicGetString, **281**
SalPicSet, 61
SalPicSetString, **279**
SalPostMsg, 55, 146
SalQueryFieldEdit, 54, **277**
SalQuit, 59
SalReportCreate, 196
SalReportPrint, **218**, 242
SalReportTableCreate, 196
SalReportTablePrint, 197
SalReportTableView, 197
SalReportView, **219**, 242
SalSendClassMessageNamed, **256**
SalSendMsg, 55, 83, 141
SalSendMsgToChildren, **137**, 144
SalSetFieldEdit, 54, **278**
SalSetWindowText, 242
SalStatusSetText, 135, **142**
SalStrCompress, **282**
SalStrGetBufferLength, **82**
SalStrLength, 81, **82**
SalStrLop, 126
SalStrToNumber, 217
SalStrUncompress, **282**
SalStrUpperX, **176**
SalTblAnyRows, 184

SalTblDefineRowHeader, 174
SalTblDoDeletes, 61, **191**
SalTblDoInserts, 61, **191**
SalTblDoUpdates, 61, 192
SalTblFetchRow, 173, 176
SalTblInsertRow, **188**
SalTblPopulate, **167**, 185
SalTblQueryFocus, 177, **178**
SalTblSetContext, 173, 176, **179**
SalTblSetFlagsAnyRows, **187**
SalTblSetFocusCell, **188**
SalTblSetRow, **171**, 173
SalTrackPopupMenu, **275**
SalVBXAddItem, 292
SalVBXCloseChannel, 292
SalVBXGetError, 292
SalVBXGetNumParam, 293
SalVBXRefresh, 293
SalVBXRemoveItem, 293
SalVBXSetPicture, 293
SalVBXSetProp, 293
SalWaitCursor, 46, 135
SAM, 47
SAM_ DragCanAutoStart, 288
SAM_Activate, 114, 144, 151
SAM_AnyEdit, 55, 81, **82**
SAM_AppExit, 42, 54, 128, 144
SAM_AppStartup, 40, **42**, 112, 128
SAM_Click, 98, 114, 132, 138, **174**, **215**
SAM_Close, **59**, 98, 115, 141, 144, 156
SAM_Create, **44**, 55, 81, 98, 112, 150, 166
SAM_CreateComplete, **98**
SAM_Destroy, 128, **144**, 156
SAM_DragDrop, 289
SAM_DragEnd, 288
SAM_DragEnter, 288
SAM_DragExit, 289
SAM_DragMove, 288
SAM_DragNotify, 288
SAM_DragStart, 288
SAM_DropFiles, 286
SAM_FetchRow, 169, 173
SAM_FetchRowDone, 169
SAM_ReportFetchInit, **221**, 224
SAM_ReportFetchNext, 224
SAM_ReportFinish, **222**, 224
SAM_ReportStart, **221**, 224
SAM_RowHeaderClick, 114, 173, **174**
SAM_RowValidate, **175**
SAM_SetFocus, 81, **83**
SAM_SqlError, 41, **46**, 128, 129, 213
SAM_User, **120**
SAM_Validate, 54, 56
SCRAPBK.APP, 264
ScrapBook, 264
SCRAPBOOK table, 265
screen, 111
SDK, 120
security, 107
SELECT, 86
selected row, **171**
selector tool, 206
sequential numbers, 37, 72
server application, 264
shared lock, 51, 72
shared Sql Handles, **77**
ShowWindow, 115, **139**
slow animate, **32**
Solo edition, 3
sql handle, **42**, 86
sql handle class, 86
SQL Server, 2
SQL.INI, 35, 52
SQLBase, 2, 51, 62, 92
SqlCommit, 73
SqlConnect, 46, 73
SQLConsole, 2
SqlDatabase, **46**, 73, 112
SQLERROR_Reason, **48**
SQLERROR_Remedy, **48**

SqlErrorText, **47**
SqlExecute, 61, 73
SqlExtractArgs, **47**
SqlFetchNext, 52, 61, 63, 64, 73
SqlFetchPrevious, 52, 61, 63, 64
SqlFetchRow, 61, 62, 63
SqlGetParameter, 51
SqlGetResultSetCount, **65**, 93
SqlGetRollbackFlag, **47**, 127
SQLHANDL.APL, 104, 116, 122
SqlImmediate, **73**
SQLNetwork, 2, **8**
SqlOpen, 61
SqlPassword, **46**, 73, 112
SqlPrepare, **61**, 73
SqlPrepareAndExecute, 61
SQLPROD, 52
SqlRetrieve, 61
SqlSetIsolationLevel, 53
SqlSetParameter, 51, 127
SQLTalk, 2, 111, 265
SQLTalk for Windows, 2
SqlUser, **46**, 73, 112
SQLWindows, 2, **3**, 109
SQLWindows Application Language, 6
SQLWindows compiler, 7, **32**
standard, 161
Starter edition, 3
statistics, 202, 229
status bar, 112, 142
storage methods, 301
StrIFF, **234**
string, 82
StrLength, **234**
struct, 119
structPointer, 119
sum, 202
summary statistics, 229
suppress blank line, 208
supressing blank lines, **235**
SW_HIDE, 140
SW_MINIMIZE, 140
SW_RESTORE, 140
SW_SHOW, 140
SW_SHOWMAXIMIZED, 140
SW_SHOWMINIMIZED, 140
SW_SHOWMINNOACTIVE, 140
SW_SHOWNA, 140
SW_SHOWNOACTIVATE, 140
SW_SHOWNORMAL, 141
SWCSTRUC.DLL, 119
SWDEMO, 37
SYSADM, 18
system menu, 142
system modal dialog box, 43

—T—

tab order, 80
table window, 110, 137, **159**
table window cache, **161**
tabs, 231
TBL_Error, 188
TBL_FillAll, **168**
TBL_FillAllBackground, **168**
TBL_FillNormal, **168**
TBL_Flag_EditLeftJustify, **169**
TBL_Flag_GrayedHeaders, **170**
TBL_Flag_HScrollByCols, **170**
TBL_Flag_MovableCols, **170**
TBL_Flag_SelectableCols, **170**
TBL_Flag_ShowVScroll, **170**
TBL_Flag_ShowWaitCursor, **170**
TBL_Flag_SingleSelection, **170**
TBL_Flag_SizableCols, **170**
TBL_Flag_SuppressLastColLine, **170**
TBL_Flag_SuppressRowLines, **170**
TBL_MaxRow, 188
TBL_MinSplitRow, 188
TBL_NoMoreRows, 174

TBL_RowDeleted, 174
TBL_RowFetched, 174
TBL_SetFirstRow, 171
TBL_SetLastRow, 171
TBL_SetNextRow, 171
TBL_SetPrevRow, 171
team programming, 7, **295**
TeamWindows, **295**
templates, 303
tester, 299
TIE, 1
tile to parent, 262
title, 138
tool bar, 37
tool palette, **11**, 83
tool palette, ReportWindows, **206**
toolbar, 112
top-level table window, 159
top-level window, 44
TPM_CenterAlign, **275**
TPM_CursorX, **276**
TPM_CursorY, **276**
TPM_LeftAlign, **276**
TPM_LeftButton, **276**
TPM_RightAlign, **276**
TPM_RightButton, **276**
transaction, 52
transparent color, **102**
two-pass totals, 201
TYPE_DataField, 70
TYPE_PushButton, 70
TYPE_RadioButton, 70

—U—

update, 37, 70, 72, 73
user interface, 107
user mode, **20**
USER.EXE, 116, 117

—V—

VBX, 117
VBX control, 6
VBX controls, **290**
views, **27**
Visual Basic control, 6
Visual Basic controls, 117
visual class, 97
Visual Toolchest, 4
visualizer, 13, 23

—W—

When SqlError, 47, **92**
window parameters, **43**
Window_Invalid, 139
Window_Maximized, 139
Window_Minimized, 139
Window_Normal, 139
Window_NotVisible, 139
Windows API, 117
WINDOWS.H, 120
WinTalk, 2, 111, 265
WM_NEXTDLGCTL, 81
WM_RBUTTONDOWN, 275, 287
WM_USER, 81
word wrap, **48**, 161
wParam, 47, 114

Rajesh Lalwani

Rajesh Lalwani has done Master of Technology (Computer Science) from the Indian Institute of Technology, New Delhi and MS (Computer Science) from the Pennsylvania State University. In addition, Lalwani has a certificate in Databases from the Stanford University.

Before coming to Gupta Corporation, Lalwani worked at Hewlett-Packard. While at Hewlett-Packard, Lalwani spoke at several Hewlett-Packard users conferences in USA, Germany, and Denmark. At Gupta, Lalwani works with the Technical Services. He presented a paper on object-oriented programming at the Gupta Developers Conference, 1993.

Lalwani welcomes any feedback and opportunities for presenting a technical talk on SQLWindows. He can be contacted by email at lalwani@aol.com, rlalwani@gupta.com, or CompuServe 73353,1065, or by FAX at (408) 446-2563.